FUJITSU SEMICONDUCTOR
CONTROLLER MANUAL

CM71-00101-5E

FR Family

32-BIT MICROCONTROLLER

INSTRUCTION MANUAL

FUJITSU

FR Family

32-BIT MICROCONTROLLER

INSTRUCTION MANUAL

FUJITSU LIMITED

PREFACE

■ Objectives and intended reader

The FR* family CPU core features proprietary Fujitsu architecture and is designed for controller applications using 32-bit RISC based computing. The architecture is optimized for use in microcontroller CPU cores for built-in control applications where high-speed control is required.

This manual is written for engineers involved in the development of products using the FR family of microcontrollers. It is designed specifically for programmers working in assembly language for use with FR family assemblers, and describes the various instructions used with FR family. Be sure to read the entire manual carefully.

Note* that the use or non-use of coprocessors, as well as coprocessor specifications depends on the functions of individual FR family products.

For information about coprocessor specifications, users should consult the coprocessor section of the product documentation. Also, for the rules of assembly language grammar and the use of assembler programs, refer to the "FR Family Assembler Manual".

* : FR, the abbreviation of FUJITSU RISC controller, is a line of products of FUJITSU Limited.

■ Trademark

The company names and brand names herein are the trademarks or registered trademarks of their respective owners.

■ Organization of this manual

This manual consists of the following 7 chapters and 1 appendix:

CHAPTER 1 FR FAMILY OVERVIEW

This chapter describes the features of the FR FAMILY CPU core, and provides sample configurations.

CHAPTER 2 MEMORY ARCHITECTURE

This chapter describes memory space in the FR family CPU.

CHAPTER 3 REGISTER DESCRIPTIONS

This chapter describes the registers used in the FR family CPU.

CHAPTER 4 RESET AND "EIT" PROCESSING

This chapter describes reset and "EIT" processing in the FR family CPU.

CHAPTER 5 PRECAUTIONARY INFORMATION FOR THE FR FAMILY CPU

This chapter presents precautionary information related to the use of the FR family CPU.

CHAPTER 6 INSTRUCTION OVERVIEW

This chapter presents an overview of the instructions used with the FR family CPU.

CHAPTER 7 DETAILED EXECUTION INSTRUCTIONS

This chapter presents each of the execution instructions used by the FR family assembler, in reference format.

APPENDIX

The appendix section includes lists of CPU instructions used in the FR family, as well as instruction map diagrams.

CONTENTS

Main changes in this edition

Page	Changes (For details, refer to main body.)
-	Be sure to refer to the "Check Sheet" for the latest cautions on development. is changed. ("Check Sheet" is seen at the following support page... is deleted.)
i	"■ Objectives and intended reader" is changed. ("FR" → "FR*")
	"■ Objectives and intended reader" is changed. (" *: " is added.)
	"PREFACE" is changed. ("■ Trademark" is added.)
	"PREFACE" is changed. ("The company names and brand names herein are the trademarks or registered trademarks of their respective owners." is added.)
9	"Table 2.1-1 Structure of a Vector Table Area" is changed. For 3F8H, ("No" → "Yes")
18	"● Lowest Bit Value of Program Counter" is changed. ("incremented by one, and therefore" → "incremented and therefore")
20	"Figure 3.3-4 "ILM" Register Functions" is changed. (A line from ILM to COMP is added.)
23	"Figure 3.3-7 Sample of Table Base Register (TBR) Operation" is changed. ("31" → "bit31")
27	"■ System Stack Pointer (SSP), User Stack Pointer (USP)" is changed. ("ST R13", "@-R15" → "ST R13, @-R15")
	The title of "Figure 3.3-12 Example of Stack Pointer Operation in Execution of Instruction "ST R13", "@-R15" when "S" Flag = 0" is changed. ("ST R13", "@-R15" → "ST R13, @-R15")
28	The title of "Figure 3.3-13 Example of Stack Pointer Operation in Execution of Instruction "ST R13", "@-R15" when "S" Flag = 1" is changed. ("ST R13", "@-R15" → "ST R13, @-R15")
28	"■ Recovery from EIT handler" is changed. ("4.2 Basic Operations in "EIT" Processing ■ Recovery from EIT handler" → "■ Recovery from EIT handler"of "4.2 Basic Operations in "EIT" Processing")
37	"4.3 Interrupts" is changed. ("External" → "User")
	"■ Sources of Interrupts" is changed. ("External" → "User")

Page	Changes (For details, refer to main body.)
38	"4.3.1 User Interrupts" is changed. ("External" → "User"), ("external" → "user")
	"■ Overview of User Interrupts" is changed. ("External" → "User")
	"■ Overview of User Interrupts" is changed. ("Interrupts are referred to as "external" when they originate outside the CPU." is deleted.)
	"■ Conditions for Acceptance of User Interrupt Requests" is changed. ("External" → "User")
	"■ Conditions for Acceptance of User Interrupt Requests" is changed. ("The CPU accepts interrupts" → "The CPU accepts user interrupts")
	"■ Operation Following Acceptance of an User Interrupt" is changed. ("External" → "User"), ("external" → "user")
39	"■ How to Use User Interrupts" is changed. ("External" → "User"), ("external" → "user")
	"Figure 4.3-1 How to Use User Interrupts" is changed. ("External" → "User")
51	"Table 4.6-1 Priority of "EIT" Requests" is changed. ("External" → "User"), ("INT" → "INTE")
62	"■ Examples of Programing Delayed Branching Instructions" is changed. (The position of comment ";not satisfy" is changed.) (R12 → R13)
66	"● Calculations are designated by a mnemonic placed between operand 1 and operand 2, with the results stored at operand 2" is changed. (The position of R2 is changed.)
72	"7.1 ADD (Add Word Data of Source Register to Destination Register)" is changed. ("Instruction bit pattern : 1010 0110 0010 0011" is added.)
75	"7.4 ADDC (Add Word Data of Source Register and Carry Bit to Destination Register)" is changed. ("Instruction bit pattern : 1010 0111 0010 0011" is added.)
79	"7.8 SUB (Subtract Word Data in Source Register from Destination Register)" is changed. ("Instruction bit pattern : 1010 1100 0010 0011" is added.)
80	"7.9 SUBC (Subtract Word Data in Source Register and Carry Bit from Destination Register)" is changed. ("Instruction bit pattern : 1010 1101 0010 0011" is added.)
81	"7.10 SUBN (Subtract Word Data in Source Register from Destination Register)" is changed. ("Instruction bit pattern : 1010 1110 0010 0011" is added.)
82	"7.11 CMP (Compare Word Data in Source Register and Destination Register)" is changed. ("Instruction bit pattern : 1010 1010 0010 0011" is added.)
85	"7.14 AND (And Word Data of Source Register to Destination Register)" is changed. ("Instruction bit pattern : 1000 0010 0010 0011" is added.)

Page	Changes (For details, refer to main body.)
87	"7.15 AND (And Word Data of Source Register to Data in Memory)" is changed. ("Instruction bit pattern : 1000 0100 0010 0011" is added.)
89	"7.16 ANDH (And Half-word Data of Source Register to Data in Memory)" is changed. ("Instruction bit pattern : 1000 0101 0010 0011" is added.)
91	"7.17 ANDB (And Byte Data of Source Register to Data in Memory)" is changed. ("Instruction bit pattern : 1000 0110 0010 0011" is added.)
92	"7.18 OR (Or Word Data of Source Register to Destination Register)" is changed. ("Instruction bit pattern : 1001 0010 0010 0011" is added.)
94	"7.19 OR (Or Word Data of Source Register to Data in Memory)" is changed. ("Instruction bit pattern : 1001 0100 0010 0011" is added.)
96	"7.20 ORH (Or Half-word Data of Source Register to Data in Memory)" is changed. ("Instruction bit pattern : 1001 0101 0010 0011" is added.)
98	"7.21 ORB (Or Byte Data of Source Register to Data in Memory)" is changed. ("Instruction bit pattern : 1001 0110 0010 0011" is added.)
99	"7.22 EOR (Exclusive Or Word Data of Source Register to Destination Register)" is changed. ("Instruction bit pattern : 1001 1010 0010 0011" is added.)
101	"7.23 EOR (Exclusive Or Word Data of Source Register to Data in Memory)" is changed. ("Instruction bit pattern : 1001 1100 0010 0011" is added.)
103	"7.24 EORH (Exclusive Or Half-word Data of Source Register to Data in Memory)" is changed. ("Instruction bit pattern : 1001 1101 0010 0011" is added.)
105	"7.25 EORB (Exclusive Or Byte Data of Source Register to Data in Memory)" is changed. ("Instruction bit pattern : 1001 1110 0010 0011" is added.)
121	"7.34 MUL (Multiply Word Data)" is changed. ("Instruction bit pattern : 1010 1111 0010 0011" is added.)
123	"7.35 MULU (Multiply Unsigned Word Data)" is changed. ("Instruction bit pattern : 1010 1011 0010 0011" is added.)
125	"7.36 MULH (Multiply Half-word Data)" is changed. ("Instruction bit pattern : 1011 1111 0010 0011" is added.)
127	"7.37 MULUH (Multiply Unsigned Half-word Data)" is changed. ("Instruction bit pattern : 1011 1011 0010 0011" is added.)
129	"7.38 DIV0S (Initial Setting Up for Signed Division)" is changed. ("Instruction bit pattern : 1001 0111 0100 0010" is added.)
131	"7.39 DIV0U (Initial Setting Up for Unsigned Division)147/308" is changed. ("Instruction bit pattern : 1001 0111 0101 0010" is added.)
133	"7.40 DIV1 (Main Process of Division)" is changed. ("Instruction bit pattern : 1001 0111 0110 0010" is added.)
135	"7.41 DIV2 (Correction when Remainder is 0)" is changed. ("Instruction bit pattern : 1001 0111 0111 0010" is added.)

Page	Changes (For details, refer to main body.)
136	"7.42 DIV3 (Correction when Remainder is 0)" is changed. ("Instruction bit pattern : 1001 1111 0110 0000" is added.)
137	"7.43 DIV4S (Correction Answer for Signed Division)" is changed. ("Instruction bit pattern : 1001 1111 0111 0000" is added.)
138	"7.44 LSL (Logical Shift to the Left Direction)" is changed. ("Instruction bit pattern : 1011 0110 0010 0011" is added.)
141	"7.47 LSR (Logical Shift to the Right Direction)" is changed. ("Instruction bit pattern : 1011 0010 0010 0011" is added.)
144	"7.50 ASR (Arithmetic Shift to the Right Direction)" is changed. ("Instruction bit pattern : 1011 1010 0010 0011" is added.)
147	"7.53 LDI:32 (Load Immediate 32-bit Data to Destination Register)" is changed. ("Instruction bit pattern : 1001 1111 1000 0011 : 1000 0111 0110 0101 : 0100 0011 0010 0001" is added.)
148	"7.54 LDI:20 (Load Immediate 20-bit Data to Destination Register)" is changed. ("Instruction bit pattern : 1001 1011 0101 0011 : 0100 0011 0010 0001" is added.)
149	"7.55 LDI:8 (Load Immediate 8-bit Data to Destination Register)" is changed. ("Instruction bit pattern : 1100 0010 0001 0011" is added.)
150	"7.56 LD (Load Word Data in Memory to Register)"is changed. ("Instruction bit pattern : 0000 0100 0010 0011" is added.)
151	"7.57 LD (Load Word Data in Memory to Register)" is changed. ("Instruction bit pattern : 0000 0000 0010 0011" is added.)
153	"7.59 LD (Load Word Data in Memory to Register)" is changed. ("o4" → "u4")
154	"7.60 LD (Load Word Data in Memory to Register)" is changed. ("Instruction bit pattern : 0000 0111 0000 0011" is added.)
156	"7.61 LD (Load Word Data in Memory to Register)" is changed. ("Instruction bit pattern : 0000 0111 1000 0100" is added.)
157	"7.62 LD (Load Word Data in Memory to Program Status Register)" is changed. Flag change: ("Ri" → "R15")
158	"7.62 LD (Load Word Data in Memory to Program Status Register)" is changed. ("Instruction bit pattern : 0000 0111 1001 0000" is added.)
159	"7.63 LDUH (Load Half-word Data in Memory to Register)" is changed. ("Instruction bit pattern : 0000 0101 0010 0011" is added.)
160	"7.64 LDUH (Load Half-word Data in Memory to Register)" is changed. ("Instruction bit pattern : 0000 0001 0010 0011" is added.)
162	"7.66 LDUB (Load Byte Data in Memory to Register)" is changed. ("Instruction bit pattern : 0000 0110 0010 0011" is added.)

Page	Changes (For details, refer to main body.)
163	"7.67 LDUB (Load Byte Data in Memory to Register)" is changed. ("Instruction bit pattern : 0000 0010 0010 0011" is added.)
165	"7.69 ST (Store Word Data in Register to Memory)" is changed. ("Instruction bit pattern : 0001 0100 0010 0011" is added.)
166	"7.70 ST (Store Word Data in Register to Memory)" is changed. ("Instruction bit pattern : 0001 0000 0010 0011" is added.)
168	"7.72 ST (Store Word Data in Register to Memory)" is changed. ("o4" → "u4")
169	"7.73 ST (Store Word Data in Register to Memory)" is changed. ("Instruction bit pattern : 0001 0111 0000 0011" is added.)
170	"7.74 ST (Store Word Data in Register to Memory)" is changed. ("Instruction bit pattern : 0001 0111 1000 0100" is added.)
171	"7.75 ST (Store Word Data in Program Status Register to Memory)" is changed. ("Instruction bit pattern : 0001 0111 1001 0000" is added.)
172	"7.76 STH (Store Half-word Data in Register to Memory)" is changed. ("Instruction bit pattern : 0001 0101 0010 0011" is added.)
173	"7.77 STH (Store Half-word Data in Register to Memory)" is changed. ("Instruction bit pattern : 0001 0001 0010 0011" is added.)
175	"7.79 STB (Store Byte Data in Register to Memory)" is changed. ("Instruction bit pattern : 0001 0110 0010 0011" is added.)
176	"7.80 STB (Store Byte Data in Register to Memory)" is changed. ("Instruction bit pattern : 0001 0010 0010 0011" is added.)
178	"7.82 MOV (Move Word Data in Source Register to Destination Register)" is changed. ("Instruction bit pattern : 1000 1011 0010 0011" is added.)
179	"7.83 MOV (Move Word Data in Source Register to Destination Register)" is changed. ("Instruction bit pattern : 1011 0111 0101 0011" is added.)
180	"7.84 MOV (Move Word Data in Program Status Register to Destination Register)" is changed. ("Instruction bit pattern : 0001 0111 0001 0011" is added.)
181	"7.85 MOV (Move Word Data in Source Register to Destination Register)" is changed. ("Instruction bit pattern : 1011 0011 0101 0011" is added.)
183	"7.86 MOV (Move Word Data in Source Register to Program Status Register)" is changed. ("Instruction bit pattern : 0000 0111 0001 0011" is added.)
184	"7.87 JMP (Jump)" is changed. ("Instruction bit pattern : 1001 0111 0000 0001" is added.)

Page	Changes (For details, refer to main body.)
185	"7.88 CALL (Call Subroutine)" is changed. ("extension for use as the branch destination address" → "extension")
	"7.88 CALL (Call Subroutine)" is changed. ("CALL 120H" → " CALL label ... label: ; CALL instruction address + 122_H")
	"7.88 CALL (Call Subroutine)" is changed. ("Instruction bit pattern : 1101 0000 1001 0000" is added.)
186	"7.89 CALL (Call Subroutine)" is changed. ("Instruction bit pattern : 1001 0111 0001 0001" is added.)
187	"7.90 RET (Return from Subroutine)" is changed. ("Instruction bit pattern : 1001 0111 0010 0000" is added.)
188	"7.91 INT (Software Interrupt)" is changed. ("INT#9" to "#13", "#64", "#65" → "INT#9" to "INT#13", "INT#64", "INT#65")
189	"7.91 INT (Software Interrupt)" is changed. ("Instruction bit pattern : 0001 1111 0010 0000" is added.)
191	"7.92 INTE (Software Interrupt for Emulator)" is changed. ("Instruction bit pattern : 1001 1111 0011 0000") is added.
192	"7.93 RETI (Return from Interrupt)" is changed. (D2, D1, → S,)
193	"7.93 RETI (Return from Interrupt)" is changed. ("Instruction bit pattern : 1001 0111 0011 0000" is added.)
194	"7.94 Bcc (Branch Relative if Condition Satisfied)" is changed. ("extension, for use as the branch destination address." → "extension")
195	"7.94 Bcc (Branch Relative if Condition Satisfied)" is changed. ("BHI 50H" → " BHI label ... label: ; BHI instruction address + 50_H")
196	"7.95 JMP:D (Jump)" is changed. ("Instruction bit pattern : 1001 1111 0000 0001" is added.)
197	"7.96 CALL:D (Call Subroutine)" is changed. ("extension for use as the branch destination address" → "extension")

Page	Changes (For details, refer to main body.)
198	"7.96 CALL:D (Call Subroutine)" is changed. ("CALL : D 120H LDI : 8 #0, R2 ; Instruction placed in delay slot" → "CALL:D label LDI : 8 #0, R2 ; Instruction placed in delay slot ... label: ; CALL: D instruction address + 122_H")
	"7.96 CALL:D (Call Subroutine)" is changed. ("Instruction bit pattern : 1101 1000 1001 0000" is added.)
200	"7.97 CALL:D (Call Subroutine)" is changed. ("Instruction bit pattern : 1001 1111 0001 0001" is added.)
202	"7.98 RET:D (Return from Subroutine)" is changed. ("Instruction bit pattern : 1001 1111 0010 0000" is added.)
203	"7.99 Bcc:D (Branch Relative if Condition Satisfied)" is changed. ("extension, for use as the branch destination address" → "extension")
204	"7.99 Bcc:D (Branch Relative if Condition Satisfied)" is changed. ("BHI :D 50H LDI :8 #255, R1 ; Instruction placed in delay slot" → "BHI:D label ... LDI :8 #255, R1 ; Instruction placed in delay slot label: ; BHI: D instruction address + 50_H")
	"7.99 Bcc:D (Branch Relative if Condition Satisfied)" is changed. ("Instruction bit pattern : 1111 1111 0010 1000" is changed.)
227	"7.114 LDRES (Load Word Data in Memory to Resource)" is changed. ("Instruction bit pattern : 1011 1100 1000 0010" is added.)
228	"7.115 STRES (Store Word Data in Resource to Memory)" is changed. ("Instruction bit pattern : 1011 1101 1000 0010" is added.)
229	"7.116 COPOP (Coprocessor Operation)" is changed. ("Resource" → "Coprocessor")
231	"7.117 COPLD (Load 32-bit Data from Register to Coprocessor Register)" is changed. ("Resource" → "Coprocessor")
233	"7.118 COPST (Store 32-bit Data from Coprocessor Register to Register)" is changed. ("Resource" → "Coprocessor")
235	"7.119 COPSV (Save 32-bit Data from Coprocessor Register to Register)" is changed. ("Resource" → "Coprocessor")
237	"7.120 NOP (No Operation)" is changed. ("Instruction bit pattern : 1001 1111 1010 0000" is addded.)

Page	Changes (For details, refer to main body.)
238	"7.121 ANDCCR (And Condition Code Register and Immediate Data)" is changed. ("Instruction bit pattern : 1000 0011 1111 1110" is added.)
239	"7.122 ORCCR (Or Condition Code Register and Immediate Data)" is changed. ("Instruction bit pattern : 1001 0011 0001 0000" is added.)
240	"7.123 STILM (Set Immediate Data to Interrupt Level Mask Register)" is changed. ("Instruction bit pattern : 1000 0111 0001 0100" is added.)
242	"7.125 EXTSB (Sign Extend from Byte Data to Word Data)" is changed. ("Instruction bit pattern : 1001 0111 1000 0001" is added.)
243	"7.126 EXTUB (Unsign Extend from Byte Data to Word Data)" is changed. ("Instruction bit pattern : 1001 0111 1001 0001" is changed.)
244	"7.127 EXTSH (Sign Extend from Byte Data to Word Data)" is changed. ("Instruction bit pattern : 1001 0111 1010 0001" is added.)
245	"7.128 EXTUH (Unsigned Extend from Byte Data to Word Data)" is changed. ("Instruction bit pattern : 1001 0111 1011 0001" is added.)
255	"7.133 ENTER (Enter Function)" is changed. ("XXXX XXXX 0000 0011" → "0000 1111 0000 0011")
257	"7.134 LEAVE (Leave Function)" is changed. ("Instruction bit pattern : 1001 1111 1001 0000" is addded.)
258	"7.135 XCHB (Exchange Byte Data)" is chenged. ("extu (Rj) → Ri" → "extu ((Rj)) → Ri")
259	"7.135 XCHB (Exchange Byte Data)" is chenged. ("Instruction bit pattern : 1000 1010 0001 0000" is added.)

Page	Changes (For details, refer to main body.)
263	"A.1 Symbols Used in Instruction Lists" is chenged. ● Symbols in Mnemonic and Operation Columns is changed. i8 ....("128 to 255" → "0 to 255")
	"A.1 Symbols Used in Instruction Lists" is chenged. ● Symbols in Mnemonic and Operation Columns is changed. ("Note: Data from -128 to -1 is handled as data from 128 to 255." is deleted.)
	"A.1 Symbols Used in Instruction Lists" is chenged. ● Symbols in Mnemonic and Operation Columns is changed. i20 ....("$0x80000_H$ to $0xFFFFF_H$" → "00000_H to $FFFFF_H$")
	"A.1 Symbols Used in Instruction Lists" is chenged. ● Symbols in Mnemonic and Operation Columns is changed. ("Note: Data from $-0x80000_H$ to -1 is handled as data from $0x80000_H$ to $0xFFFFF_H$." is deleted.)
	"A.1 Symbols Used in Instruction Lists" is chenged. ● Symbols in Mnemonic and Operation Columns is changed. i32 ....("$0x80000000_H$ to $0xFFFFFFFF_H$" → "00000000_H to $FFFFFFFF_H$")
	"A.1 Symbols Used in Instruction Lists" is chenged. ● Symbols in Mnemonic and Operation Columns is changed. ("Note: Data from $-0x80000000_H$ to -1 is handled as data from $0x80000000_H$ to $0xFFFFFFFF_H$." is deleted.)
263	"A.1 Symbols Used in Instruction Lists" is changed. ● Symbols in Mnemonic and Operation Columns is changed. ("• Ri" → "• Ri, Rj")
	"● Symbols in Operation Column" is changed. ("• ( )............ indicates indirect addressing, which values reading or loading from/to the memory address where the registers within ( ) or the formula indicate. • { }............ indicates the calculation priority; ( ) is used for specifying indiiirect address" is added.)
264	"● Cycle (CYC) Column" is changed. ("special" → "dedicated")
266	"Table A.2-4 Bit Operation Instructions (8 Instructions)" is changed. ("(Ri)&=(F0H+u4)" → "(Ri)&={F0H+u4}")
	"Table A.2-4 Bit Operation Instructions (8 Instructions)" is changed. ("(Ri)&=((u4<<4)+FH)" → "(Ri)&={{u4<<4}+FH}")
	"Table A.2-4 Bit Operation Instructions (8 Instructions)" is changed. ("(Ri) | = (u4<<4)" → "(Ri) | = {u4<<4}")
	"Table A.2-4 Bit Operation Instructions (8 Instructions)" is changed. ("(Ri) ^ = (u4<<4)" → "(Ri) ^ = {u4<<4}")
	"Table A.2-4 Bit Operation Instructions (8 Instructions)" is changed. ("(Ri) & (u4<<4)" → "(Ri) & {u4<<4}")

Page	Changes (For details, refer to main body.)
267	"Table A.2-6 Shift Instructions (9 Instructions)" is changed. ("Ri <<(u4+16) → Ri" → "Ri <<{u4+16} → Ri") ("Ri >>(u4+16) → Ri" → "Ri >>{u4+16} → Ri") ("Ri >>(u4+16) → Ri" → "Ri >>{u4+16} → Ri")
272	"Table A.2-13 Direct Addressing Instructions (14 Instructions)" is changed. ("disp8" → "dir8"), ("disp9" → "dir9"), ("disp10" → "dir10")
273	"Table A.2-16 Other Instructions (16 Instructions)" is changed. ("i8" → "u8")
276	"Table B.2-1 "E" Format" is changed. ("- : Undefined" is added.)

CHAPTER 1

FR FAMILY OVERVIEW

This chapter describes the features of the FR FAMILY CPU core, and provides sample configurations.

1.1 Features of the FR Family CPU Core

The FR family CPU core features proprietary Fujitsu architecture and is designed for controller applications using 32-bit "RISC" based computing. The architecture is optimized for use in microcontroller CPU cores for built-in control applications where high-speed control is required.

■ Features of the FR Family CPU Core

- General-purpose register architecture
- Linear space for 32-bit (4 Gbytes) addressing
- 16-bit fixed instruction length (excluding immediate data, coprocessor instructions)
- 5-stage pipeline configuration for basic instructions, one-instruction one-cycle execution
- 32-bit by 32-bit computation enables completion of multiplication instructions within five cycles
- Stepwise division instructions enable 32-bit/ 32-bit division
- Direct addressing instructions for peripheral circuit access
- Coprocessor instructions for direct designation of peripheral accelerator
- High speed interrupt processing complete within 6 cycles

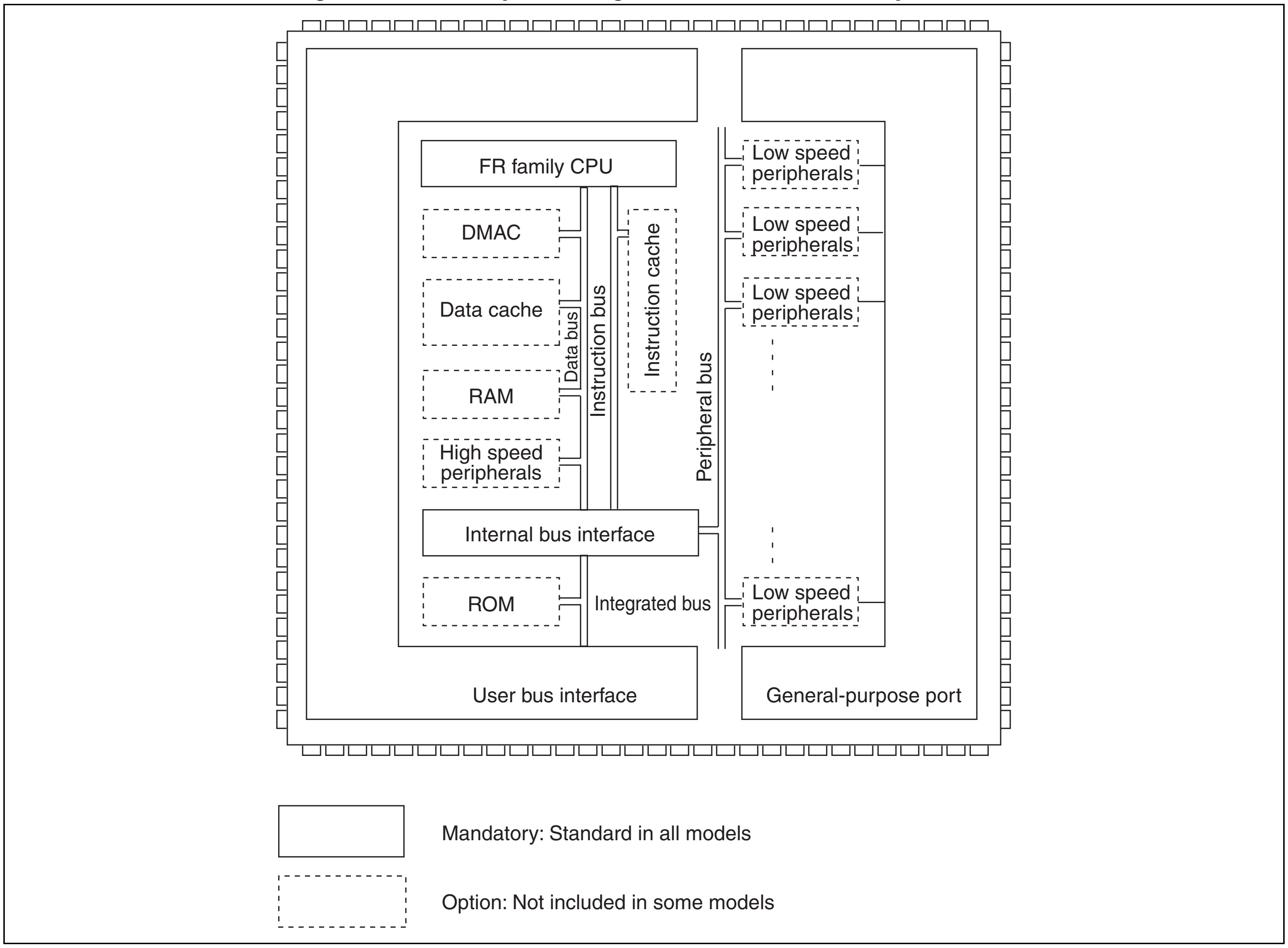
1.2 Sample Configuration of an FR Family Device

FR family devices have block configuration with bus connections between individual modules. This enables module connections to be altered as necessary to accommodate a wide variety of functional configurations.

Figure 1.2-1 shows an example of the configuration of an FR family device.

■ Sample Configuration of an FR Family Device

Figure 1.2-1 Sample Configuration of an FR Family Device

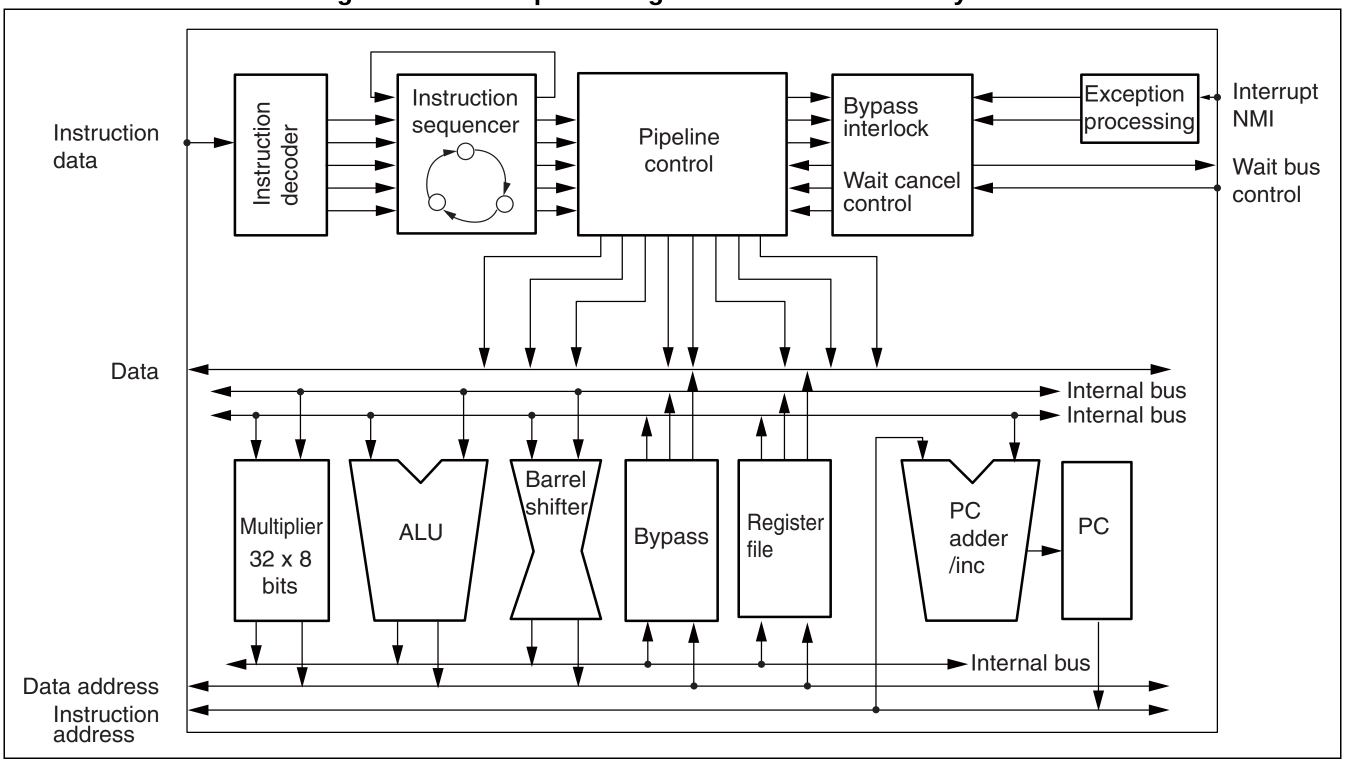


1.3 Sample Configuration of the FR Family CPU

The FR family CPU core features a block configuration organized around general-purpose registers, with dedicated registers, "ALU" units, multipliers and other features included for each specific application.
Figure 1.3-1 shows a sample configuration of an FR family CPU.

■ Sample Configuration of the FR Family CPU

Figure 1.3-1 Sample Configuration of the FR Family CPU



CHAPTER 2
MEMORY ARCHITECTURE

This chapter describes memory space in the FR family CPU.
Memory architecture includes the allocation of memory space as well as methods used to access memory.

2.1 FR Family Memory Space

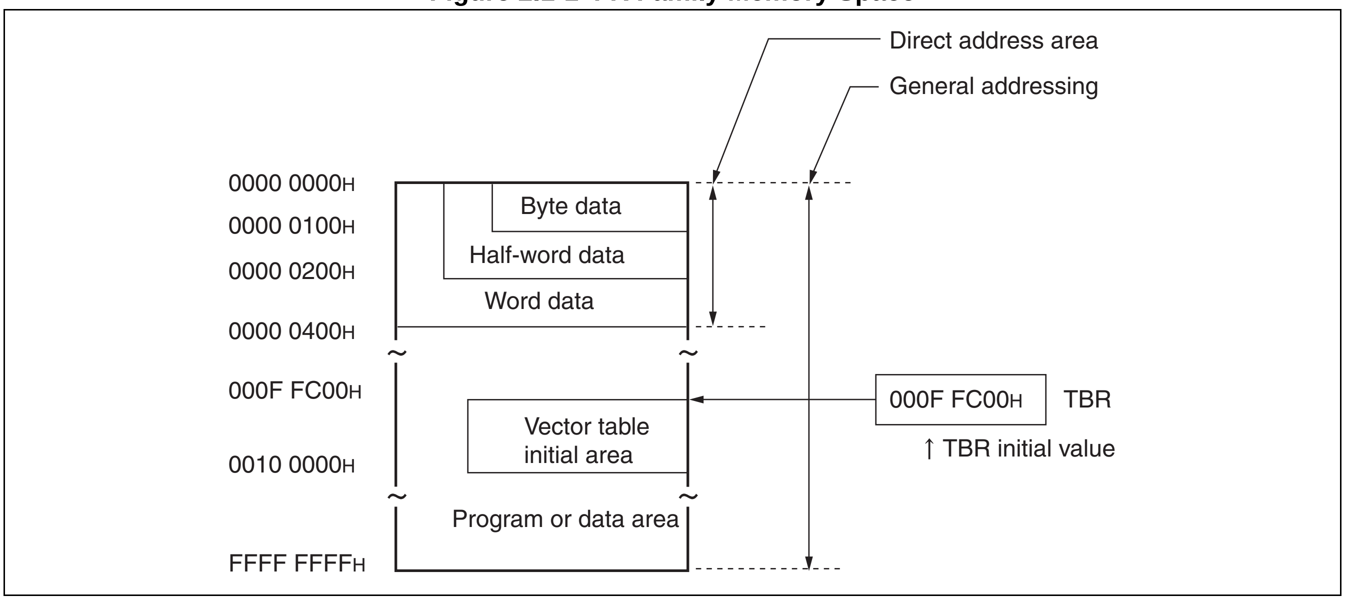
The FR family controls memory space in byte units, and provides linear designation of 32-bit spaces. Also, to enhance instruction efficiency, specific areas of memory are allocated for use as direct address areas and vector table areas.

■ Memory Space

Figure 2.1-1 illustrates memory space in the FR family.

For a detailed description of the direct address area, see Section "2.1.1 Direct Address Area", and for the vector table area, see Section "2.1.2 Vector Table Area".

Figure 2.1-1 FR Family Memory Space



■ Unused Vector Table Area

Unused vector table area is available for use as program or data area.

2.1.1 Direct Address Area

The lower portion of the address space is used for the direct address area. Instructions that specify direct addresses allow you to access this area without the use of general-purpose registers, using only the operand information in the instruction itself. The size of the address area that can be specified by direct addressing varies according to the length of the data being transferred.

■ Direct Address Area

The size of the address area that can be specified by direct addressing varies according to the length of the data being transferred, as follows:

- Transfer of byte data: 0000 0000_H to 0000 $00FF_H$
- Transfer of half-word data: 0000 0000_H to 0000 $01FF_H$
- Transfer of word data: 0000 0000_H to 0000 $03FF_H$

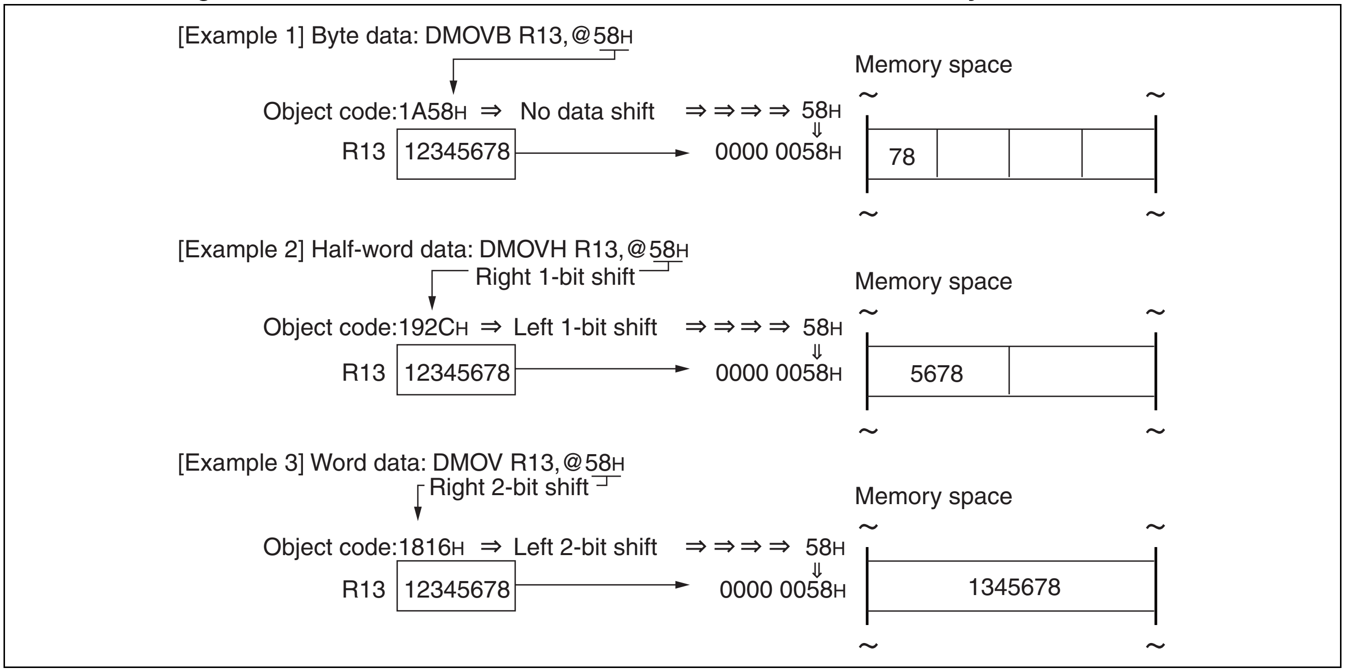
■ Use of Operand Information Contained in Instructions

The 8-bit address information contained in the instruction has the following significance.

- In byte data: Value represents the lower 8 bits of the address.
- In half-word data: Value is doubled and used as the lower 9 bits of the address.
- In word data: Value is multiplied by 4 and used as the lower 10 bits of the address.

Figure 2.1-2 shows the relationship between the length of the data that designates the direct address, and the actual address in memory.

Figure 2.1-2 Relation between Direct Address Data and Memory Address Value



2.1.2 Vector Table Area

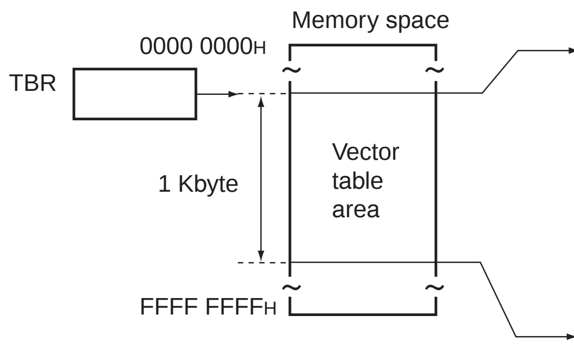
An area of 1 Kbyte beginning with the address shown in the table base register (TBR) is used to store "EIT" vector addresses.

■ Overview of Vector Table Areas

An area of 1 Kbyte beginning with the address shown in the table base register (TBR) is used to store "EIT" vector addresses. Data written to this area includes entry addresses for exception processing, interrupt processing and trap processing.

The table base register (TBR) can be rewritten to allocate this area to any desired location within word alignment limitations.

Figure 2.1-3 Relation between Table Base Register (TBR) and Vector Table Addresses



Number	Offset from TBR	EIT source
FFH	000H	Entry address for INT instruction
FEH	004H	Entry address for INT instruction
FDH	008H	Entry address for INT instruction
FCH	00CH	Entry address for INT instruction
00H	3FCH	Entry address for reset processing

■ Contents of Vector Table Areas

A vector table is composed of entry addresses for each of the "EIT" processing programs. Each table contains some values whose use is fixed according to the CPU architecture, and some that vary according to the types of built-in peripheral circuits present. Table 2.1-1 shows the structure of a vector table area.

Table 2.1-1 Structure of a Vector Table Area

Offset from TBR	Number (hex)	Model-dependent	EIT value description	Remarks
000_H	FF_H	No	INT #$0FF_H$	
004_H	FE_H	No	INT #$0FE_H$	
~	~	~	~	~
$2F8_H$	41_H	No	System reserved	Do not use
$2FC_H$	40_H	No	System reserved	
~	~	~	~	~
$33C_H$	30_H	No	INT #030_H	
340_H	$2F_H$	Yes	INT #$02F_H$ or IR31	Values will increase towards higher limits when using over 32-source extension. Refer to User's Manual for each model.
344_H	$2E_H$	Yes	INT #$02E_H$ or IR30	
~	~	~	~	
$3BC_H$	10_H	Yes	INT #010_H or IR00	
$3C0_H$	$0F_H$	No	INT #$00F_H$ or NMI	
$3C4_H$	$0E_H$	No	Undefined instruction exception	
$3C8_H$	$0D_H$	No	Emulator exception	
$3CC_H$	$0C_H$	No	Step trace break trap	
$3D0_H$	$0B_H$	No	Operand break trap	
$3D4_H$	$0A_H$	No	Instruction break trap	
$3D8_H$	09_H	No	Emulator exception	
$3DC_H$	08_H	No	INT #008_H or coprocessor error trap	
$3E0_H$	07_H	No	INT #007_H or coprocessor not-found trap	
$3E4_H$	06_H	No	System reserved	Do not use
~	~	~	~	~
$3F8_H$	01_H	Yes	System reserved or Mode Vector	Refer to User's Manual for each model.
$3FC_H$	00_H	No	Reset	*

*: Even when the "TBR" value is changed, the reset vector remains the fixed address "$000FFFFC_H$".

■ Vector Table Area Initial Value

After a reset, the value of the table base register (TBR) is initialized to "$000FFC00_H$", so that the vector table area is between addresses "$000FFC00_H$" and "$000FFFFF_H$".

2.2 Bit Order and Byte Order

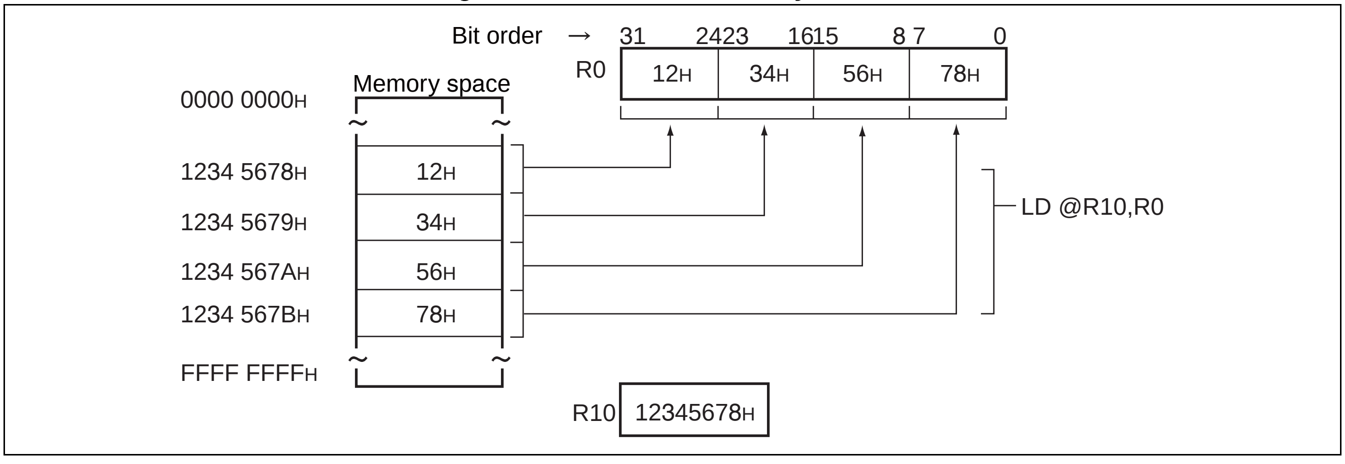
This section describes the order in which three types of data, 8, 16, and 32 bits, are placed in the memory in the FR family.
In the FR family, the bit number increases approaching the MSB, and the byte number increases approaching the lowest address value.

■ Bit Order and Byte Order

Bit order in the general-purpose register is that the larger numbers are placed in the vicinity of the MSB while the smaller numbers are near the LSB. Byte order configuration requires the upper data to be placed in the smaller address memory, while the lower data are placed in the larger address memory.

Figure 2.2-1 illustrates the bit order and byte order in the FR family.

Figure 2.2-1 Bit Order and Byte Order



2.3 Word Alignment

In the FR family, the type of data length used determines restrictions on the designation of memory addresses (word alignment).

■ Program Restrictions on Word Alignment

When using half-word instruction length, memory addresses must be accessed in multiples of two. With branching instructions and other instructions that may result in attempting to store odd numbered values to the "PC", the lowest value in the "PC" will be read as "0". Thus an even numbered address will always be generated by fetching a branching instruction.

■ Data Restrictions on Word Alignment

● Word data

Data must be assigned to addresses that are multiples of 4. Even if the operand value is not a multiple of 4, the lower two bits of the memory address will explicitly be read as "0".

● Half-word data

Data must be assigned to addresses that are multiples of 2. Even if the operand value is not a multiple of 2, the lowest bit of the memory address will explicitly be read as "0".

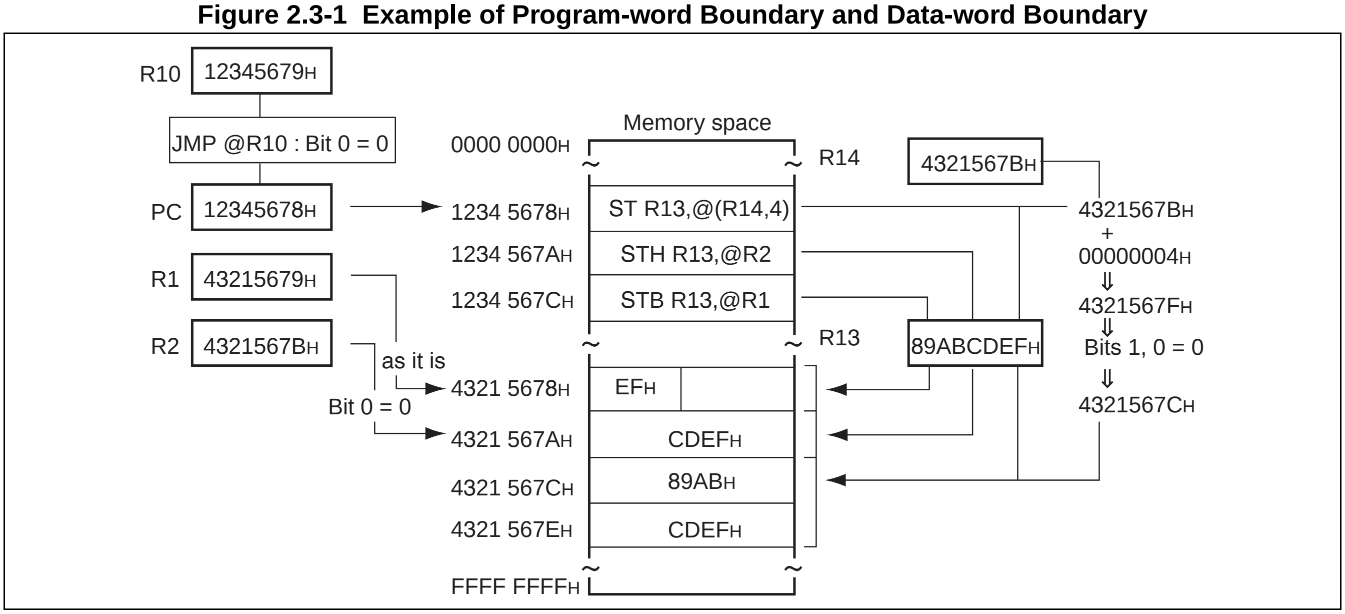
● Byte data

There are no restrictions on addresses.

The forced setting of some bits to "0" during memory access for word data and half-word data is applied after the computation of the execution address, not at the source of the address information.

Figure 2.3-1 shows an example of the program-word boundary and data-word boundary.

Figure 2.3-1 Example of Program-word Boundary and Data-word Boundary



CHAPTER 3
REGISTER DESCRIPTIONS

This chapter describes the registers used in the FR family CPU.

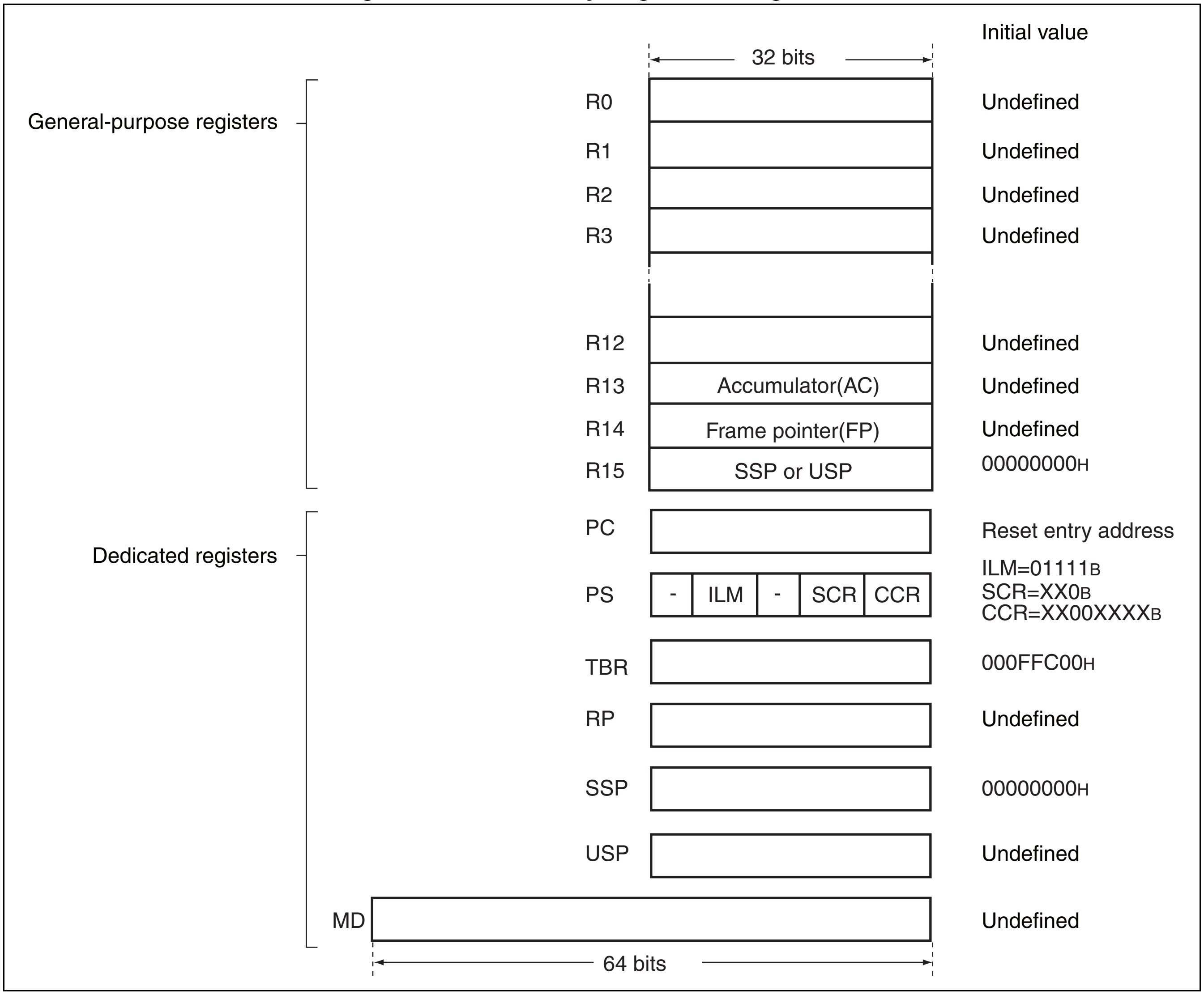
3.1 FR Family Register Configuration

FR family devices use two types of registers, general-purpose registers and dedicated registers.
- **General-purpose registers: Store computation data and address information**
- **Dedicated registers: Store information for specific applications**

Figure 3.1-1 shows the configuration of registers in FR family devices.

■ FR Family Register Configuration

Figure 3.1-1 FR Family Register Configuration



3.2 General-purpose Registers

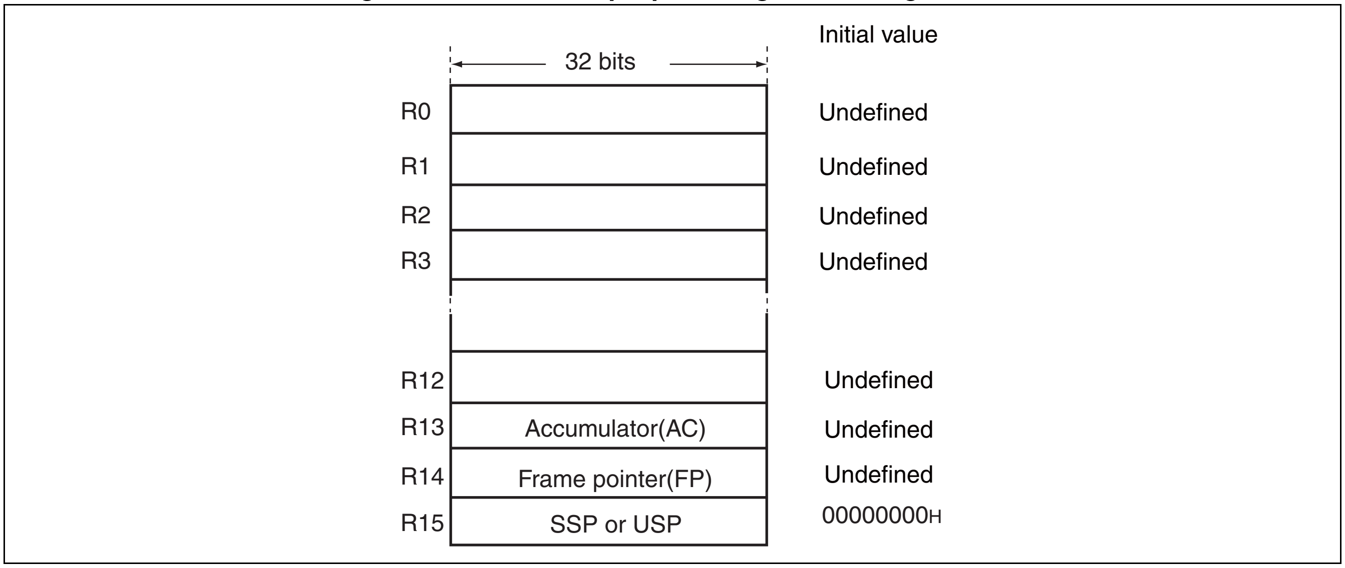
The FR family CPU uses general-purpose registers to hold the results of various calculations, as well as information about addresses to be used as pointers for memory access. These registers also have special functions with certain types of instructions.

■ Overview of General-purpose Registers

The FR family CPU has sixteen (16) general-purpose registers each 32 bits in length. Normal instructions can use any of these sixteen registers without distinction.

Figure 3.2-1 shows the configuration of a general-purpose register.

Figure 3.2-1 General-purpose Register Configuration



■ Special Uses of General-purpose Registers

In addition to functioning as general-purpose registers, "R13", "R14", and "R15" have the following special uses with certain types of instructions.

● R13 (Accumulator: AC)

- Base address register for load/store to memory instructions
  [Example: LD @(R13, Rj), Ri]
- Accumulator for direct address designation
  [Example: DMOV @dir10,R13]
- Memory pointer for direct address designation
  [Example: DMOV @dir10, @R13+]

● R14 (Frame Pointer: FP)

- Index register for load/store to memory instructions
  [Example: LD @(R14, disp10), Ri]
- Frame pointer for reserve/release of dynamic memory area
  [Example: ENTER #u10]

● R15 (Stack Pointer: SP)

- Index register for load/store to memory instructions
  [Example: LD @(R15, udisp6), Ri]
- Stack pointer
  [Example: LD @R15+, Ri]
- Stack pointer for reserve/release of dynamic memory area
  [Example: ENTER #u10]

■ Relation between "R15" and Stack Pointer

The "R15" functions physically as either the system stack pointer (SSP) or user stack pointer (USP) for the general-purpose registers. When the notation "R15" is used in an instruction, this register will function as the "USP" if the "S" flag in the condition code register (CCR) section of the program status register (PS) is set to "1". The R15 register will function as the "SSP" if the "S" flag is set to "0".

Ensure that the S flag value is set to "0" when R15 is recovered from the EIT handler with the RETI instruction.

■ Initial Value of General-purpose Registers

After a reset, the value of registers "R00" through "R14" are undefined, and the value of "R15" is "00000000_H".

3.3 Dedicated Registers

The FR family has six 32-bit registers reserved for various special purposes, plus one 64-bit dedicated register for multiplication and division operations.

■ Dedicated Registers

The following seven dedicated registers are provided. For details, see the descriptions in Sections "3.3.1 Program Counter (PC)" through "3.3.6 Multiplication/Division Register (MD)".

● 32-bit Dedicated Registers

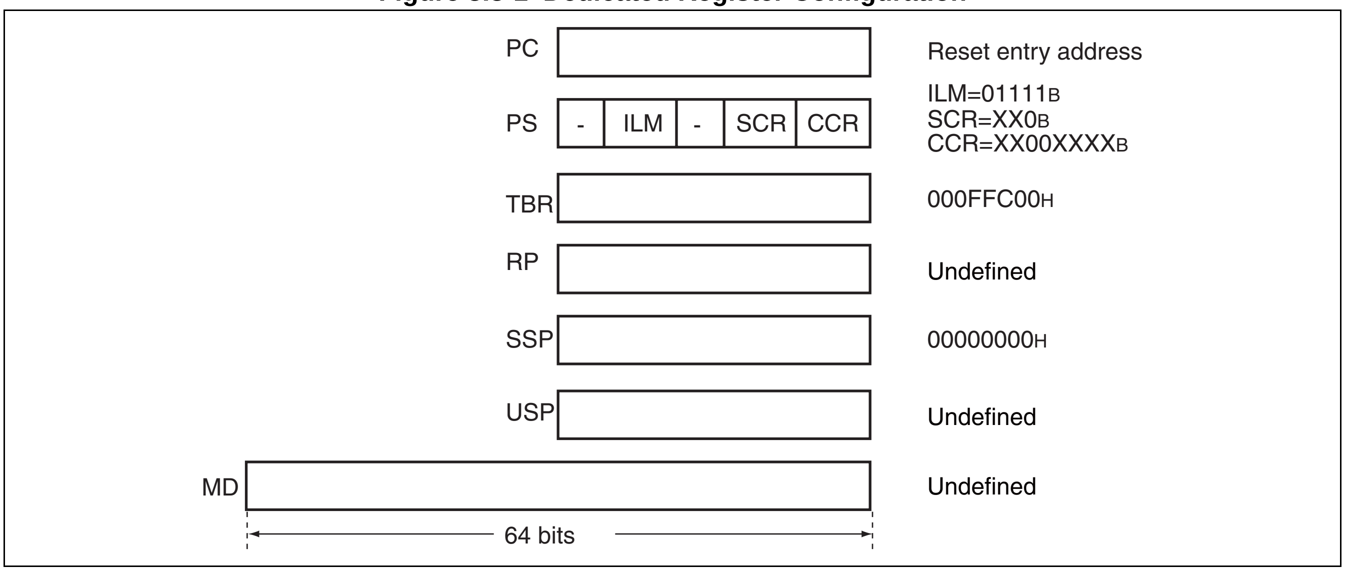
- Program counter (PC)
- Program status (PS)
- Table base register (TBR)
- Return pointer (RP)
- System stack pointer (SSP)
- User stack pointer (USP)

● 64-bit Dedicated Register

- Multiplication/Division Register (MD)

Figure 3.3-1 shows the configuration of the dedicated registers.

Figure 3.3-1 Dedicated Register Configuration



3.3.1 Program Counter (PC)

This register indicates the address containing the instruction that is currently executing. Following a reset, the contents of the PC are set to the reset entry address contained in the vector table.

■ Overview of the Program Counter

This register indicates the address containing the instruction that is currently executing. The value of the lowest bit is always read as "0", and therefore all instructions must be written to addresses that are multiples of 2.

■ Program Counter Functions

● Lowest Bit Value of Program Counter

The value of the lowest bit in the program counter is read as "0" by the internal circuits in the FR family device. Even if "1" is written to this bit, it will be treated as "0" for addressing purposes. A physical cell does exist for this bit, however, the lowest bit value remains "0" even when the program address value is incremented and therefore the value of this bit is always "0" except following a branching operation.

Because the internal circuits in the FR family device are designed to read the value of the lowest bit as "0", all instructions must be written to addresses that are multiples of 2.

● Program Counter Initial Value

Following a reset, the contents of the PC are set to the reset entry address contained in the vector table. Because initialization is applied first to the table base register (TBR), the value of the reset vector address will be "$000FFFFC_H$".

3.3.2 Program Status (PS)

The program status (PS) indicates the status of program execution, and consists of the following three parts:

- **Interrupt level mask register (ILM)**
- **System condition code register (SCR)**
- **Condition code register (CCR)**

■ Overview of Program Status Register

The program status register consists of sections that set the interrupt enable level, control the program trace break function in the CPU, and indicate the status of instruction execution.

■ Program Status Register Configuration

Figure 3.3-2 shows the configuration of the program status register.

Figure 3.3-2 Program Status Register Configuration

Bit no. →	31 ― 21	20 ― 16	15 ― 11	10 ― 08	07 ― 00
PS	Unused	ILM	Unused	SCR	CCR

■ Unused Bits in the Program Status Register

Unused bits are all reserved for future system expansion. Write values should always be "0". The read value of these bits is always "0".

■ Interrupt Level Mask Register (ILM: Bit 20 to bit 16)

● Bit Configuration of the ILM Register

Figure 3.3-3 Bit Configuration of the ILM Register

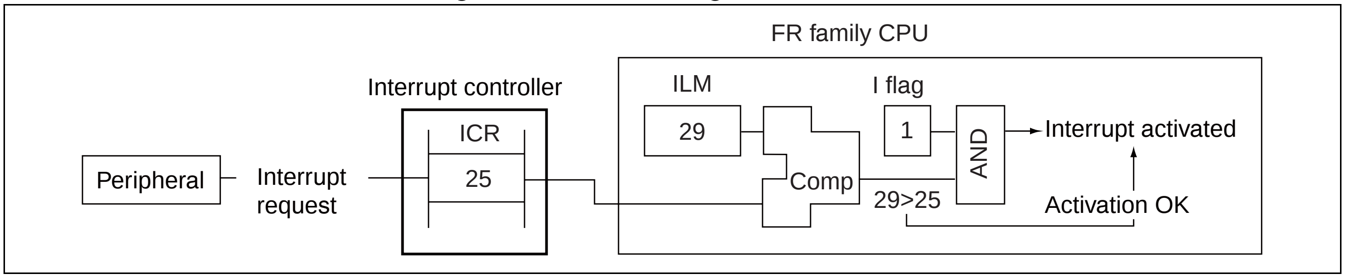
	20	19	18	17	16	
ILM	ILM4	ILM3	ILM2	ILM1	ILM0	Initial value: 01111B

● ILM Functions

The "ILM" determines the level of interrupt that will be accepted. Whenever the "I" flag in the "CCR" register is "1", the contents of this register are compared to the level of the current interrupt request. If the value of this register is greater than the level of the request, interrupt processing is activated. Interrupt levels are higher in priority at value approaching "0", and lower in priority at increasing values up to "31". Note that bit "ILM4" differs from the other bits in the register, in that setting values for this bit are restricted.

Figure 3.3-4 shows the functions of the "ILM".

Figure 3.3-4 "ILM" Register Functions



● Range of ILM Program Setting Values

If the original value of the register is in the range 16 to 31, the new value may be set in the range 16 to 31. If an instruction attempts to set a value between 0 and 15, that value will be converted to "setting value + 16" and then transferred.

If the original value is in the range 0 to 15, any new value from 0 to 31 may be set.

● Initialization of the ILM at Reset

The reset value is "01111_B".

■ System Condition Code Register (SCR: Bit 10 to bit 08)

● Bit Configuration of the SCR

Figure 3.3-5 Bit Configuration of the SCR

	10	09	08	
SCR	D1	D0	T	Initial value: $XX0_B$

● SCR Functions

- Bits D1, D0
  Bits "D1", "D0" are used for intermediate data in stepwise division calculations. This register is used to assure resumption of division calculations when the stepwise division program is interrupted during processing. If changes are made to the contents of this register during division processing, the results of the division are not assured.
- T-bit
  The T-bit is a step trace trap flag. When this bit is set to "1", step trace trap operation is enabled.
  Note: Step trace trap processing routines cannot be debugged using emulators.

● Initialization of the SCR at Reset

The values of bits "D1", "D0" are undefined, and the T-bit is set to "0".

■ Condition Code Register (CCR: Bit 07 to bit 00)

● Bit Configuration of the "CCR"

Figure 3.3-6 Bit Configuration of the "CCR"

	07	06	05	04	03	02	01	00	
CCR	-	-	S	I	N	Z	V	C	Initial value: --00XXXXB

● "CCR" Functions

- "S" Flag
  This flag selects the stack pointer to be used. The value "0" selects the system stack pointer (SSP), and "1" selects the user stack pointer (USP).
  RETI instruction is executable only when the S flag is "0".
- "I" Flag
  This flag is used to enable/disable system interrupts. The value "0" disables, and "1" enables interrupts.
- "N" Flag
  This flag is used to indicate positive or negative values when the results of a calculation are expressed in two’s complement form. The value "0" indicates positive, and "1" indicates negative.
- "Z" Flag
  This flag indicates whether the results of a calculations are zero. The value "0" indicates a non-zero value, and "1" indicates a zero value.
- "V" Flag
  This flag indicates that an overflow occurred when the results of a calculation are expressed in two’s complement form. The value "0" indicates no overflow, and "1" indicates an overflow.
- "C" Flag
  This flag indicates whether a carry or borrow condition has occurred in the highest bit of the results of a calculation. The value "0" indicates no carry or borrow, and "1" indicates a carry or borrow condition. This bit is also used with shift instructions, and contains the value of the last bit that is "shifted out".

● Initialization of the "CCR" at Reset

Following a reset, the "S" and "I" flags are set to "0" and the "N", "Z", "V" and "C" flags are undefined.

■ Note on PS Register

Because of prior processing of the PS register by some commands, a break may be brought in an interrupt processing subroutine during the use of a debugger or flag display content in the PS register may be changed with the following exceptional operations. In both cases, right re-processing is designed to execute after returning from the EIT. So, operations before and after EIT are performed conforming to the specifications.

● When a) a user interrupt or NMI is executed, b) step execution is implemented, or c) a break occurs in a data event or emulator menu due to a command just before DIV0U/DIV0S commands, the following operation may be implemented.

(1) D0 and D1 flags are changed first.
(2) EIT process routine (user interrupt, NMI or emulator) is executed.
(3) Returning from EIT, DIV0U/DIV0S commands are executed and D0 and D1 flags are set to the same value in "(1)".

● When a user interrupt or NMI factor exists, and a command such as ORCCR/STILM/ MOV Ri,PS is executed to allow an interruption, the following operation is executed:

(1) PS register is changed first.
(2) EIT process routine (user interrupt, NMI) is executed.
(3) Returning from EIT, any above command is executed and PS register is set to the same value in "(1)".

3.3.3 Table Base Register (TBR)

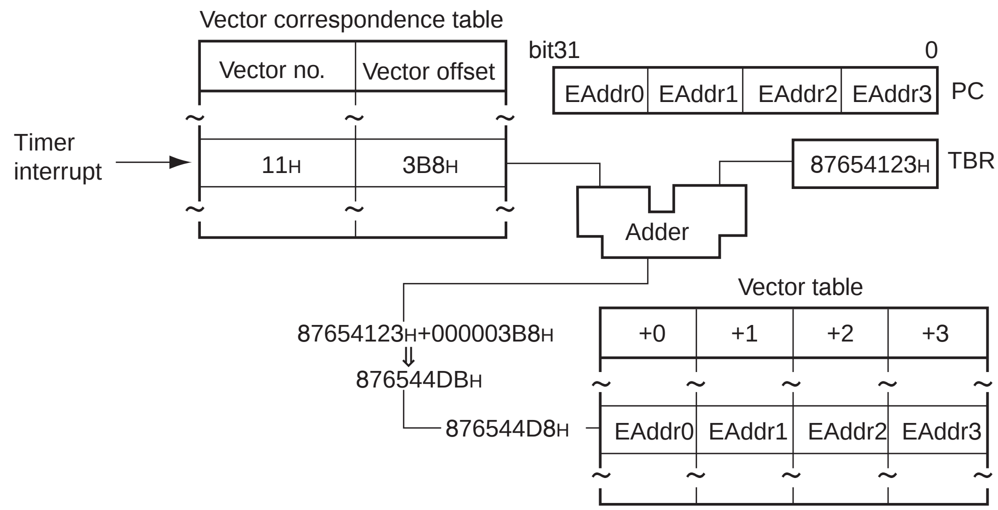
The Table Base Register (TBR) designates the table containing the entry address for "EIT" operations.

■ Overview of the Table Base Register

The Table Base Register (TBR) designates the table containing the entry address for "EIT" operations. When an "EIT" condition occurs, the address of the vector reference is determined by the sum of the contents of this register and the vector offset corresponding to the "EIT" operation.

Figure 3.3-7 shows an example of the operation of the table base register.

Figure 3.3-7 Sample of Table Base Register (TBR) Operation

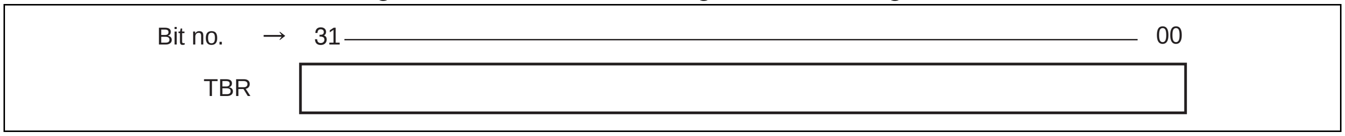


Note: The process of referencing a vector table involves application of address alignment rules for word access.

■ Table Base Register Configuration

Figure 3.3-8 shows the bit configuration of the table base register.

Figure 3.3-8 Table Base Register Bit Configuration



■ Table Base Register Functions

● Vector Table Reference Addresses

Addresses for vector reference are generated by adding the contents of the "TBR" register and the vector offset value, which is determined by the type of interrupt used. Because vector access is in word units, the lower two bits of the resulting address value are explicitly read as "0".

● Vector Table Layout

Vector table layout can be realized in word (32 bits) units.

● Initial Values in Table Base Register

After a reset, the initial value is "$000FFC00_H$".

■ Precautions Related to the Table Base Register

The "TBR" should not be assigned values greater than "$FFFFFC00_H$". If values higher than this are placed in the register, the operation may result in an overflow when summed with the offset value. An overflow condition will result in vector access to the area "00000000_H" to "$000003FF_H$", which can cause program runaway.

3.3.4 Return Pointer (RP)

The return pointer (RP) is a register used to contain the program counter (PC) value during execution of call instructions, in order to assure return to the correct address after the call instruction has executed.

■ Overview of the Return Pointer

The contents of the return pointer (RP) depend on the type of instruction. For a call instruction with a delay slot, the value is the address stored +4, and for a call instruction with no delay slot, the value is the address stored +2. The save data is returned from the "RP" pointer to the "PC" counter by execution of a "RET" instruction.

Figure 3.3-9 shows a sample operation of the "RP" pointer in the execution of a "CALL" instruction with no delay slot, and Figure 3.3-10 shows a sample operation of the "RP" pointer in the execution of a "RET" instruction.

Figure 3.3-9 Sample Operation of "RP" in Execution of a "CALL" Instruction with No Delay Slot

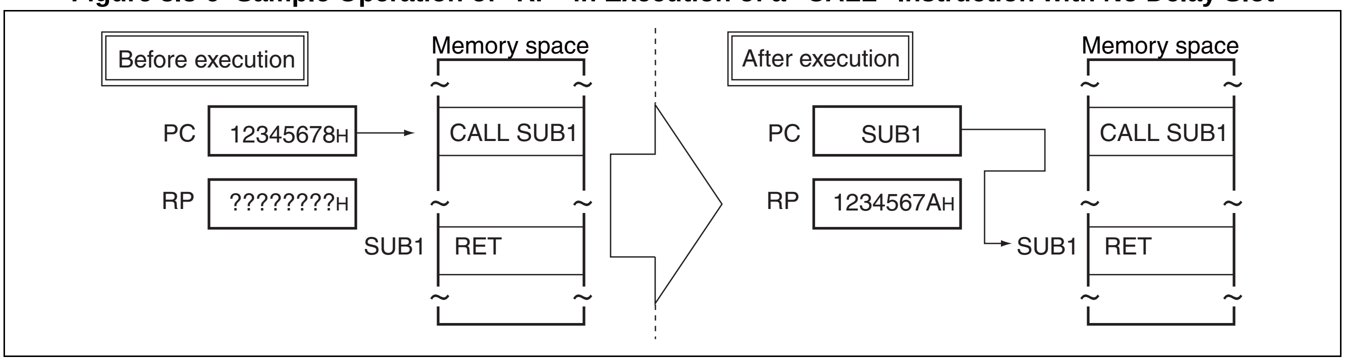
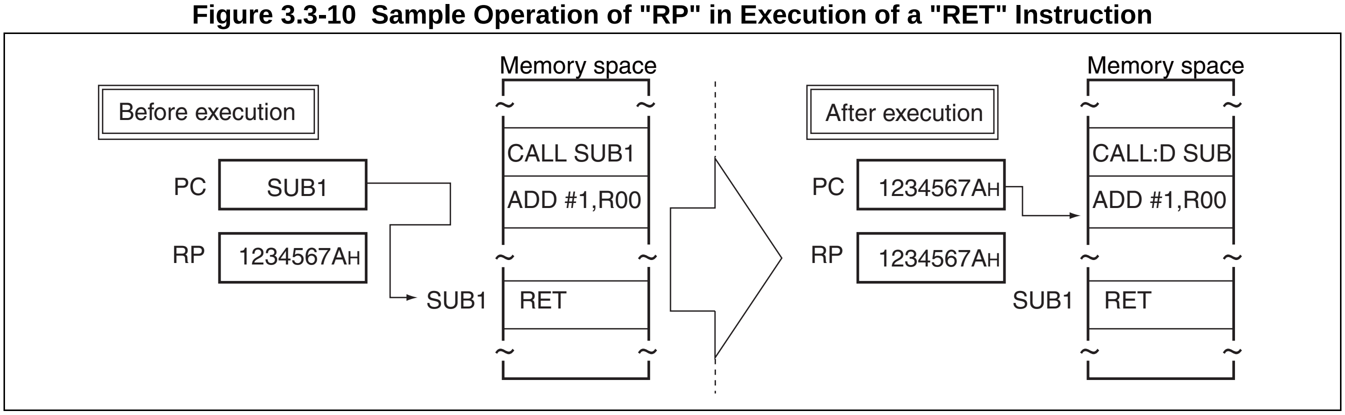


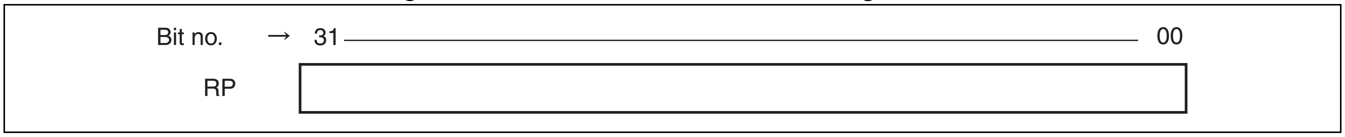
Figure 3.3-10 Sample Operation of "RP" in Execution of a "RET" Instruction



■ Return Pointer Configuration

Figure 3.3-11 shows the bit configuration of the return pointer.

Figure 3.3-11 Return Pointer Bit Configuration



■ Return Pointer Functions

● Return Pointer in Multiple "CALL" Instructions

Because the "RP" does not have a stack configuration, it is necessary to first execute a save when calling one subroutine from another subroutine.

● Initial Value of Return Pointer

The initial value is undefined.

3.3.5 System Stack Pointer (SSP), User Stack Pointer (USP)

The system stack pointer (SSP) and user stack pointer (USP) are registers that refer to the stack area. The "S" flag in the "CCR" determines whether the "SSP" or "USP" is used. Also, when an "EIT" event occurs, the program counter (PC) and program status (PS) values are saved to the stack area designated by the "SSP", regardless of the value of the "S" flag at that time.

■ System Stack Pointer (SSP), User Stack Pointer (USP)

The system stack pointer (SSP) and user stack pointer (USP) are pointers that refer to the stack area. The stack area is accessed by instructions that use general-purpose register "R15" as an indirect register, as well as register multi-transfer instructions. "R15" is used as an indirect register by the "SSP" when the "S" flag in the condition code register (CCR) is "0" and the "USP" when the "S" flag is "1". Also, when an "EIT" event occurs, the program counter (PC) and program status (PS) values are saved to the stack area designated by the "SSP", regardless of the value of the "S" flag at that time.

Figure 3.3-12 shows an example of stack pointer operation in executing the instruction "ST R13, @-R15" when the "S" flag is set to "0". Figure 3.3-13 shows an example of the same operation when the "S" flag is set to "1".

Figure 3.3-12 Example of Stack Pointer Operation in Execution of Instruction "ST R13, @-R15" when "S" Flag = 0

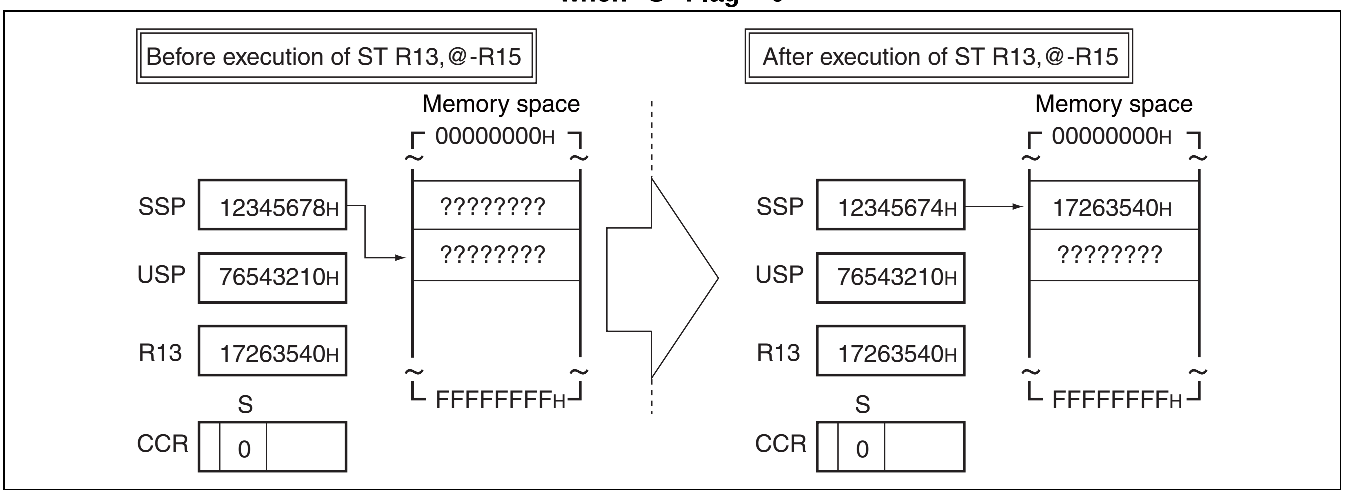
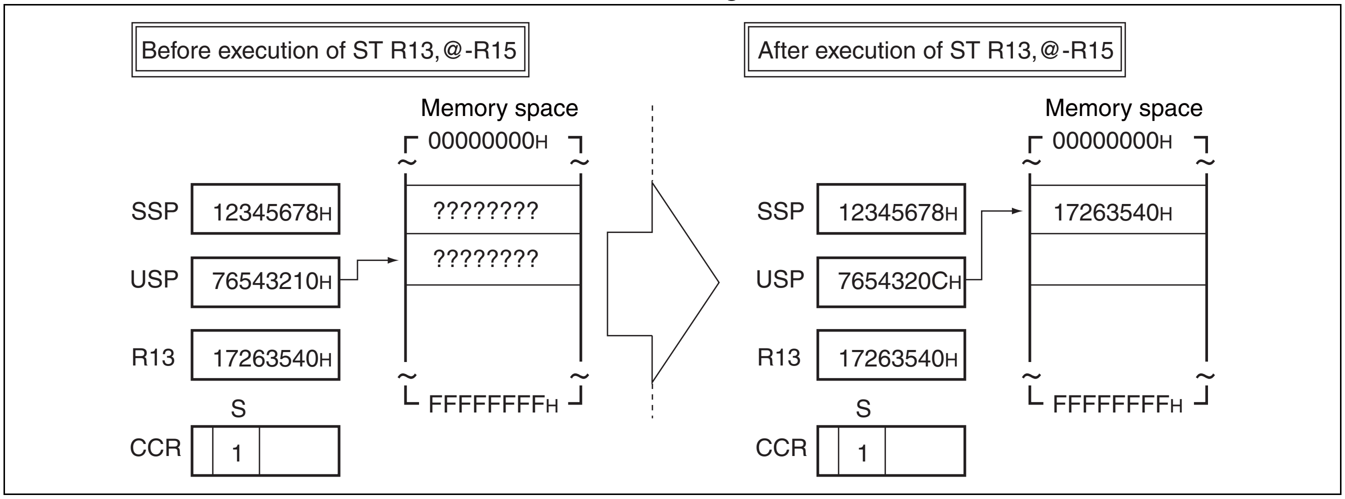


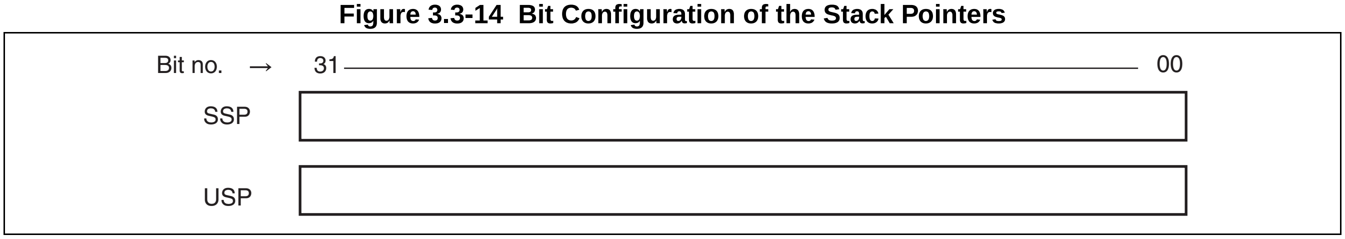
Figure 3.3-13 Example of Stack Pointer Operation in Execution of Instruction "ST R13, @-R15" when "S" Flag = 1



■ Stack Pointer Configuration

Figure 3.3-14 shows the bit configuration of the stack pointer.

Figure 3.3-14 Bit Configuration of the Stack Pointers



■ Functions of the System Stack Pointer and User Stack Pointer

● Automatic increment/decrement of stack pointer

The stack pointer uses automatic pre-decrement/post-increment counting.

● Stack Pointer Initial Value

The "SSP" has the initial value "00000000_H". The "USP" initial value is undefined.

■ Recovery from EIT handler

When RETI instruction is used for recovery from an EIT handler, it is necessary to set the "S" flag to "0" and select the system stack. For further details, see "■ Recovery from EIT handler" of "4.2 Basic Operations in "EIT" Processing".

3.3.6 Multiplication/Division Register (MD)

The multiplication/division register (MD) is a 64-bit register used to contain the result of multiplication operations, as well as the dividend and result of division operations.

■ Overview of the Multiplication/Division Register

The multiplication/division register (MD) is a register used to contain the result of multiplication operations, as well as the dividend and result of division operations. The products of multiplication are stored in the "MD" in 64-bit format. In division operations, the dividend must first be placed in the lower 32 bits of the "MD" beforehand. Then as the division process is executed, the remainder is placed in the higher 32 bits of the "MD", and the quotient in the lower 32 bits.

Figure 3.3-15 shows an example of the use of the "MD" in multiplication, and Figure 3.3-16 shows an example of division.

Figure 3.3-15 Sample Operation of "MD" in Multiplication

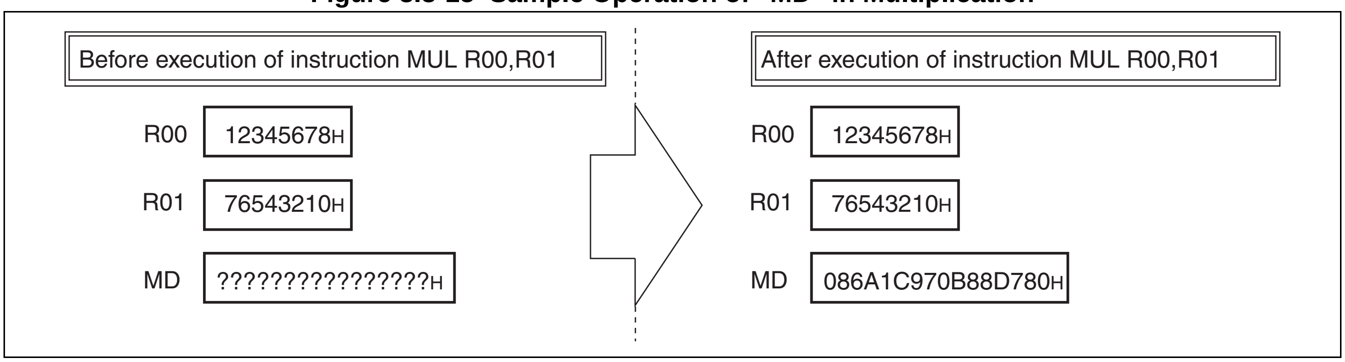
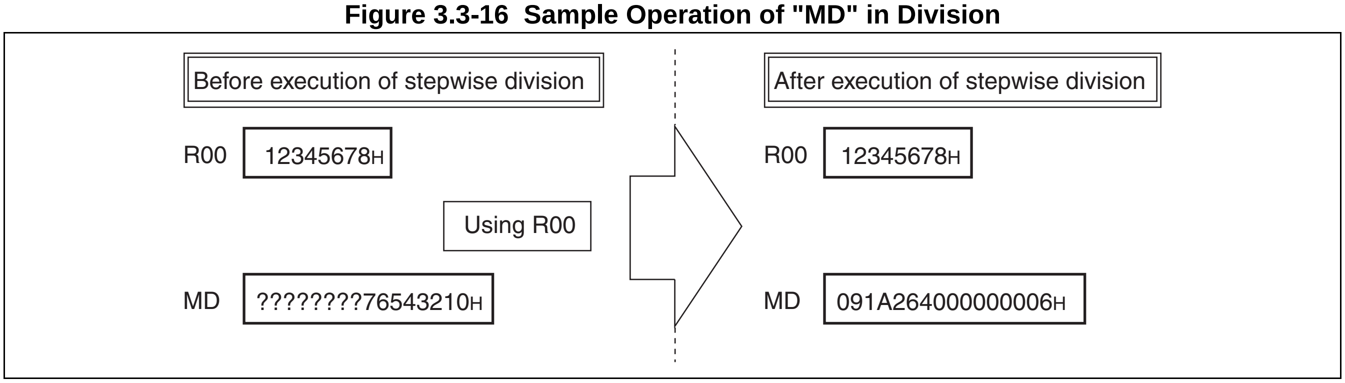


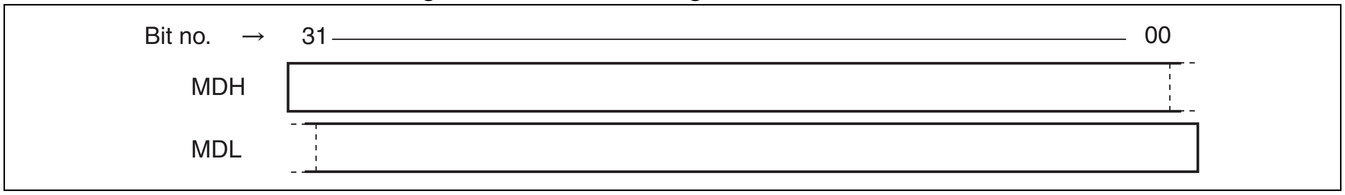
Figure 3.3-16 Sample Operation of "MD" in Division



■ Configuration of the "MD" Register

Figure 3.3-17 shows the bit configuration of the "MD".

Figure 3.3-17 Bit Configuration of the "MD"



■ Functions of the "MD"

● Storing Results of Multiplication and Division

The results of multiplication operations are stored in the "MDH" (higher 32 bits) and "MDL" (lower 32 bits) registers.

The results of division are stored as follows: quotients in the 32-bit "MDL" register, and remainders in the 32-bit "MDH" register.

● Initial Value of the "MD"

The initial value is undefined.

CHAPTER 4

RESET AND "EIT" PROCESSING

This chapter describes reset and "EIT" processing in the FR family CPU.

A reset is a means of forcibly terminating the currently executing process, initializing the entire device, and restarting the program from the beginning. "EIT" processing, in contrast, terminates the currently executing process and saves restart information to the memory, then transfers control to a predetermined processing program. "EIT" processing programs can return to the prior program by use of the "RETI" instruction.

"EIT" processing operates in essentially the same manner for exceptions, interrupts and traps, with the following minor differences.

- **Interrupts originate independently of the instruction sequence. Processing is designed to resume from the instruction immediately following the acceptance of the interrupt.**
- **Exceptions are related to the instruction sequence, and processing is designed to resume from the instruction in which the exception occurred.**
- **Traps are also related to the instruction sequence, and processing is designed to resume from the instruction immediately following the instruction in which the trap occurred.**

CHAPTER 4 RESET AND "EIT" PROCESSING

4.1 Reset Processing

A reset is a means of forcibly terminating the currently executing process, initializing the entire device, and restarting the program from the beginning. Resets are used to start the LSI operating from its initial state, as well as to recover from error conditions.

■ Reset Operations

When a reset is applied, the CPU terminates processing of the instruction executing at that time and goes into inactive status until the reset is canceled. When the reset is canceled, the CPU initializes all internal registers and starts execution beginning with the program indicated by the new value of the program counter (PC).

■ Initialization of CPU Internal Register Values at Reset

When a reset is applied, the FR family CPU initializes internal registers to the following values.

- PC: Word data stored at address "$000FFFFC_H$"
- ILM: "01111_B"
- T Flag: "0" (trace OFF)
- I Flag: "0" (interrupt disabled)
- S Flag: "0" (use SSP pointer)
- TBR: "$000FFC00_H$"
- SSP: "00000000_H"
- R00 to R14: Undefined
- R15: SSP

For a description of built-in functions following a reset, refer to the Hardware Manual provided with each FR family device.

■ Reset Priority Level

Resets have a higher priority than all "EIT" operations.

4.2 Basic Operations in "EIT" Processing

Interrupts, exceptions and traps are similar operations applied under partially differing conditions. Each "EIT" event involves terminating the execution of instructions, saving information for restarting, and branching to a designated processing program.

■ Basic Operations in "EIT" Processing

The FR family device processes "EIT" events as follows.

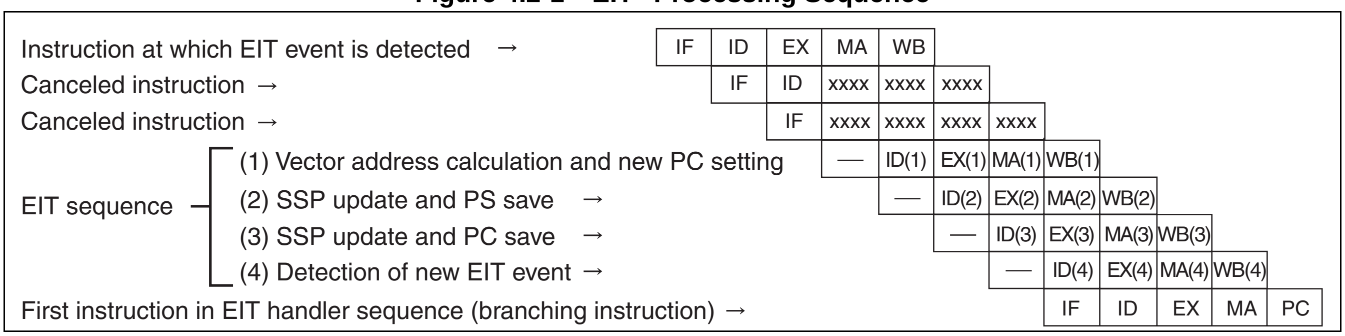
(1) The vector table indicated by the table base register (TBR) and the number corresponding to the particular "EIT" event are used to determine the entry address for the processing program for the "EIT".

(2) For restarting purposes, the contents of the old program counter (PC) and the old program status (PS) are saved to the stack area designated by the system stack pointer (SSP).

(3) After the processing flow is completed, the presence of new "EIT" sources is determined.

Figure 4.2-1 shows the operations in the "EIT" processing sequence.

Figure 4.2-1 "EIT" Processing Sequence



Note:

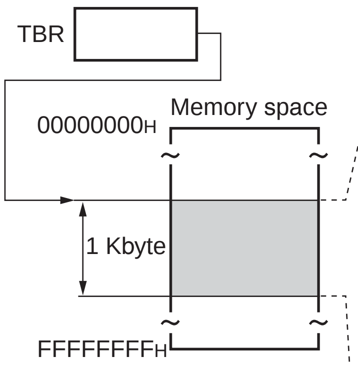
For a description of pipeline operations, see Section "5.1 Pipeline Operation".

■ Vector Table Configuration

Vector tables are located in the main memory, occupying an area of 1 Kbyte beginning with the address shown in the TBR. These areas are intended for use as a table of entry addresses for "EIT" processing, however in applications where vector tables are not required, this area can be used as a normal instruction or data area.

Figure 4.2-2 shows the structure of the vector table. (Example of 32-source)

Figure 4.2-2 Vector Table Configuration



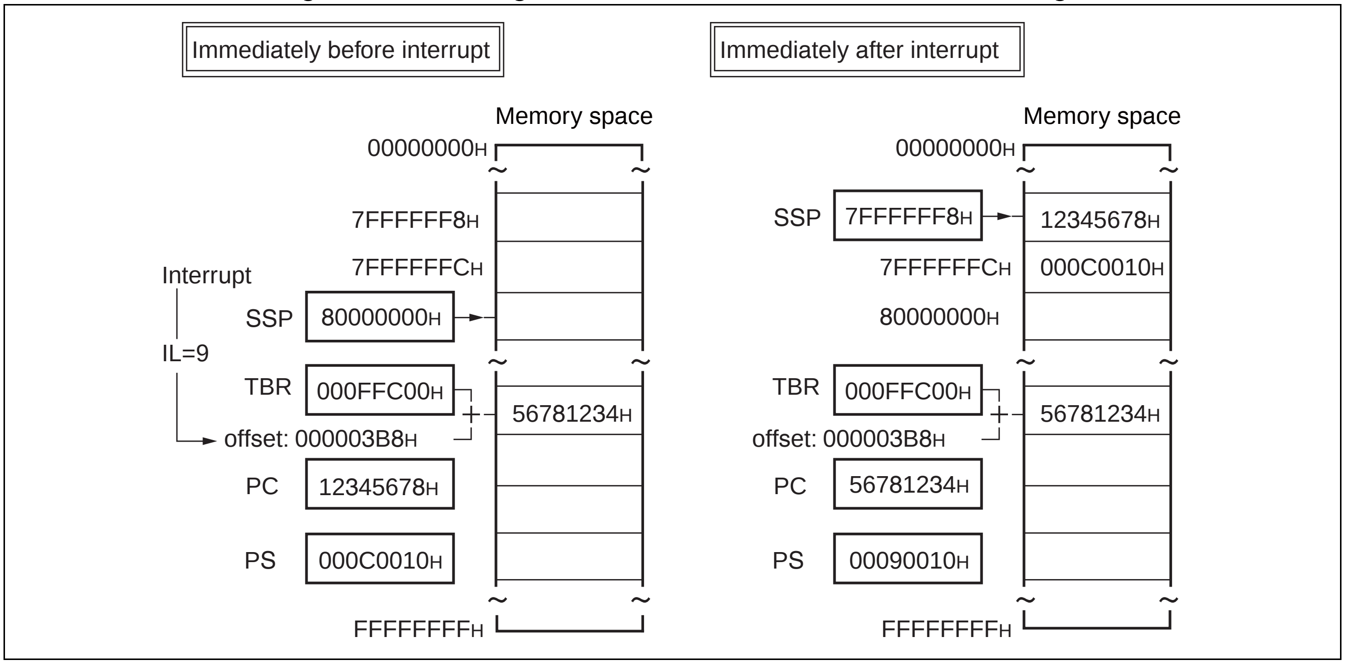
Offset	Vector no.	Description
000H	FFH	INT #0FFH
004H	FEH	INT #0FEH
008H	FDH	INT #0FDH
~	~	~
33CH	30H	INT #030H
340H	2FH	INT #02FH or IR31
344H	2EH	INT #02EH or IR30
~	~	~
3BCH	10H	INT #010H or IR00
3C0H	0FH	INT #00FH or NMI
3C4H	0EH	Undefined instruction exception
3C8H	0DH	Emulator exception
3CCH	0CH	Step trace trap
3D0H	0BH	Operand break trap
~	~	~
3F8H	01H	System reserved or Mode Vector
3FCH	00H	Reset

■ Saved Registers

Except in the case of reset processing, the values of the "PS" and "PC" are saved to the stack as designated by the "SSP", regardless of the value of the "S" flag in the "CCR". No save operation is used in reset processing.

Figure 4.2-3 illustrates the saving of the values of the "PC" and "PS" in "EIT" processing.

Figure 4.2-3 Saving "PC" and "PS" Values in "EIT" Processing



■ Recovery from EIT handler

RETI instruction is used for recovery from the EIT handler.

To insure the program execution results after recovery, it is required that all the contents of the CPU register are saved.

Ensure that the PC and PS values in the stack are not overwritten unless necessary because those values, saved in the stack at the occurrence of EIT, are recovered from the stack during the recovery sequence using the RETI instruction. Be sure to set the "S" flag to "0" when the RETI instruction is executed.

4.3 Interrupts

Interrupts originate independently of the instruction sequence. They are processed by saving the necessary information to resume the currently executing instruction sequence, and then starting the processing routine corresponding to the type of interrupt that has occurred.

There are two types of interrupt sources.

- **User interrupts**
- **Non-maskable interrupts (NMI)**

■ Overview of Interrupt Processing

Interrupts originate independently of the instruction sequence. They are processed by saving the necessary information to resume the currently executing instruction sequence, and then starting the processing routine corresponding to the type of interrupt that has occurred.

Instructions loaded and executing in the CPU before the interrupt will be executed to completion, however, any instructions loaded in the pipeline after the interrupt will be canceled. After completion of interrupt processing, therefore, execution will return to the next instruction following the generation of the interrupt signal.

■ Sources of Interrupts

There are two types of interrupt sources.

- User interrupts (See Section "4.3.1 User Interrupts")
- Non-maskable interrupts (NMI) (See Section "4.3.2 Non-maskable Interrupts (NMI)")

■ Interrupts during Execution of Stepwise Division Programs

To enable resumption of processing when interrupts occur during stepwise division programs, intermediate data is placed in the program status (PS), and saved to the stack. Therefore, if the interrupt processing program overwrites the contents of the "PS" data in the stack, the processor will resume executing the stepwise division instruction following the completion of interrupt processing, however the results of the division calculation will be incorrect.

4.3.1 User Interrupts

User interrupts originate as requests from peripheral circuits. Each interrupt request is assigned an interrupt level, and it is possible to mask requests according to their level values.

This section describes conditions for acceptance of user interrupts, as well as their operation and uses.

■ Overview of User Interrupts

User interrupts originate as requests from peripheral circuits.

Each interrupt request is assigned an interrupt level, and it is possible to mask requests according to their level values. Also, it is possible to disable all interrupts by using the I flag in the condition code register (CCR) in the program status (PS).

It is possible to enter an interrupt signal through a signal pin, but in virtually all cases the interrupt originates from the peripheral circuits contained on the FR family microcontroller chip itself.

■ Conditions for Acceptance of User Interrupt Requests

The CPU accepts user interrupts when the following conditions are met:

- The peripheral circuit is operating and generates an interrupt request.
- The interrupt enable bit in the peripheral circuit's control register is set to "enable".
- The value of the interrupt request (ICR[*1]) is lower than the value of the ILM[*2] setting.
- The "I" flag is set to "1".

  *1: ICR = Interrupt Control Register ...a register on the microcontroller that controls interrupts

  *2: ILM = Interrupt Level Mask Register ... a register in the CPU's program status (PS)

■ Operation Following Acceptance of a User Interrupt

The following operating sequence takes place after a user interrupt is accepted.

- The contents of the program status (PS) are saved to the system stack.
- The address of the next instruction is saved to the system stack.
- The value of the system stack pointer (SSP) is reduced by 8.
- The value (level) of the accepted interrupt is stored in the "ILM".
- The value "0" is written to the "S" flag in the condition code register (CCR) in the program status (PS).
- The vector address of the accepted interrupt is stored in the program counter (PC).

■ Time to Start of Interrupt Processing

The time required to start interrupt processing can be expressed as a maximum of "n + 6" cycles from the start of the instruction currently executing when the interrupt was received, where "n" represents the number of execution cycles in the instruction.

If the instruction includes memory access, or insufficient instructions are present, the corresponding number of wait cycles must be added.

■ "PC" Values Saved for Interrupts

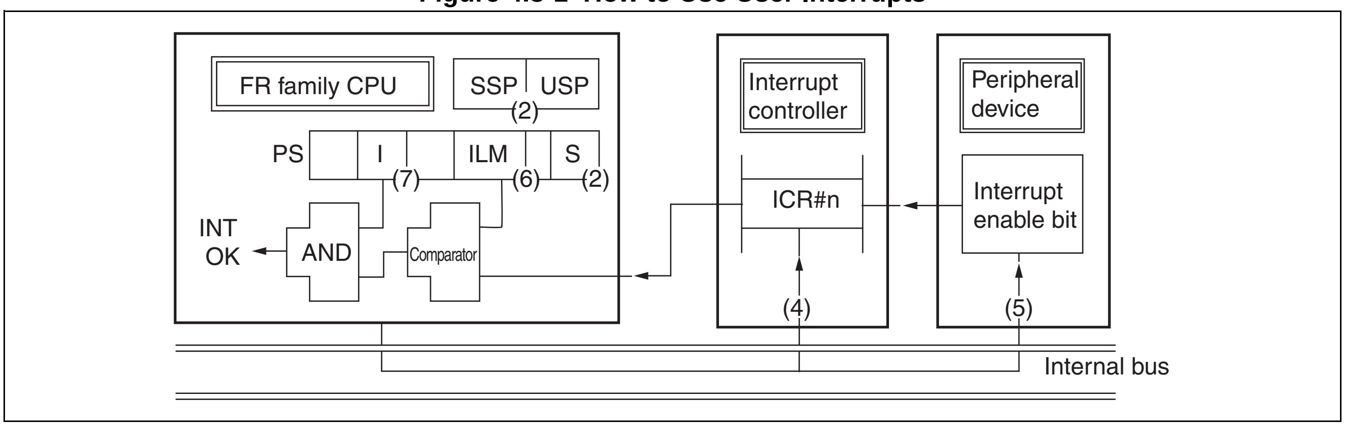
When an interrupt is accepted by the processor, those instructions in the pipeline that cannot be interrupted in time will be executed. The remainder of the instructions will be canceled, and will not be processed after the interrupt. The "EIT" processing sequence saves "PC" values to the system stack representing the addresses of canceled instructions.

■ How to Use User Interrupts

The following programming steps must be set up to enable the use of user interrupts.

Figure 4.3-1 illustrates the use of user interrupts.

Figure 4.3-1 How to Use User Interrupts



(1) Enter values in the interrupt vector table (defined as data).

(2) Set up the "SSP" values.

(3) Set up the table base register (TBR) values.

(4) Within the interrupt controller, enter the appropriate level for the "ICR" corresponding to interrupts from the peripheral from which the interrupt will originate.

(5) Initialize the peripheral function that requests the occurrence of the interrupt, and enable its interrupt function.

(6) Set up the appropriate value in the "ILM" field in the "PS".

(7) Set the "I" flag to "1".

4.3.2 Non-maskable Interrupts (NMI)

Non-maskable interrupts (NMI) are interrupts that cannot be masked. "NMI" requests can be produced when "NMI" external signal pin input to the microcontroller is active. This section describes conditions for the acceptance of "NMI" interrupts, as well as their operation and uses.

■ Overview of Non-maskable Interrupts

Non-maskable interrupts (NMI) are interrupts that cannot be masked. "NMI" requests can be produced when "NMI" external signal pin input to the microcontroller is active.

Non-maskable interrupts cannot be disabled by the "I" flag in the condition code register (CCR) in the program status (PS).

The masking function of the interrupt level mask register (ILM) in the "PS" is valid for "NMI". However, it is not possible to use the software input to set "ILM" values for masking of "NMI", so that these interrupts cannot be masked by programming.

■ Conditions for Acceptance of Non-maskable Interrupt Requests

The FR family CPU will accept an "NMI" request when the following conditions are met:

● If "NMI" Pin Input is Active:

- In normal operation: Detection of a negative signal edge
- In stop mode: Detection of an "L" level signal

● If the "ILM" Value is Greater than 15.

■ Operation Following Acceptance of a Non-maskable Interrupt

When an "NMI" is accepted, the following operations take place:

(1) The contents of the "PS" are saved to the system stack.

(2) The address of the next instruction is saved to the system stack.

(3) The value of the system stack pointer (SSP) is reduced by 8.

(4) The value "15" is written to the "ILM".

(5) The value "0" is written to the "S" flag in "CCR" in the "PS".

(6) The value "TBR + $3C0_H$" is stored in the program counter (PC).

■ Time to Start of Non-maskable Interrupt Processing

The time required to start processing of an "NMI" can be expressed as a maximum of "n + 6" cycles from the start of the instruction currently executing when the interrupt was received, where "n" represents the number of execution cycles in the instruction.

If the instruction includes memory access, or insufficient instructions are present, the corresponding number of wait cycles must be added.

■ "PC" Values Saved for Non-maskable Interrupts

When an "NMI" is accepted by the processor, those instructions in the pipeline that cannot be interrupted in time will be executed. The remainder of the instructions will be canceled, and will not be processed after the interrupt. The "EIT" processing sequence saves "PC" values to the system stack representing the addresses of canceled instructions.

■ How to Use Non-maskable Interrupts

The following programming steps must be set up to enable the use of "NMI".

(1) Enter values in the interrupt vector table (defined as data).

(2) Set up the "SSP" values.

(3) Set up "TBR" values.

(4) Set up the appropriate value in the "ILM" field in the "PS".

4.4 Exception Processing

Exceptions originate from within the instruction sequence. Exceptions are processed by first saving the necessary information to resume the currently executing instruction, and then starting the processing routine corresponding to the type of exception that has occurred.

■ Overview of Exception Processing

Exceptions originate from within the instruction sequence. Exceptions are processed by first saving the necessary information to resume the currently executing instruction, and then starting the processing routine corresponding to the type of exception that has occurred.

Branching to the exception processing routine takes place before execution of the instruction that has caused the exception.

The address of the instruction in which the exception occurs becomes the program counter (PC) value that is saved to the stack.

■ Factors Causing Exception Processing

The factor which causes the exception processing is the undefined-instruction exception (For details, see "4.4.1 Undefined Instruction Exceptions").

4.4.1 Undefined Instruction Exceptions

Undefined instruction exceptions are caused by attempts to execute instruction codes that are not defined.
This section describes the operation, time requirements and uses of undefined-instruction exceptions.

■ Overview of Undefined Instruction Exceptions

Undefined instruction exceptions are caused by attempts to execute instruction codes that are not defined.

■ Operations of Undefined Instruction Exceptions

The following operating sequence takes place when an undefined instruction exception occurs.

(1) The contents of the program status (PS) are saved to the system stack.
(2) The address of the instruction that caused the undefined-instruction exception is saved to the system stack.
(3) The value of the system stack pointer (SSP) is reduced by 8.
(4) The value "0" is written to the "S" flag in the condition code register (CCR) in the "PS".
(5) The value "TBR + $3C4_H$" is stored in the program counter (PC).

■ Time to Start of Undefined Instruction Exception Processing

The time required to start exception processing is 7 cycles.

■ "PC" Values Saved for Undefined Instruction Exceptions

The address saved to the system stack as a "PC" value represents the instruction itself that caused the undefined instruction exception. When a RETI instruction is executed, the contents of the system stack should be rewritten with the exception processing routine so that execution will either resume from the address of the next instruction after the instruction that caused the exception, or branch to the appropriate processing routine.

■ How to Use Undefined Instruction Exceptions

The following programming steps must be set up to enable the use of undefined instruction exceptions.

(1) Enter values in the interrupt vector table (defined as data).
(2) Set up the "SSP" value.
(3) Set up "TBR" value.

■ Undefined Instructions Placed in Delay Slots

Undefined instructions placed in delay slots do not generate undefined instruction exceptions. In such cases, undefined instructions have the same operation as "NOP" instructions.

4.5 Traps

Traps originate from within the instruction sequence. Traps are processed by first saving the necessary information to resume processing from the next instruction in the sequence, and then starting the processing routine corresponding to the type of trap that has occurred.

Sources of traps include the following:

- **"INT" instructions**
- **"INTE" instructions**
- **Step trace traps**
- **Coprocessor not found traps**
- **Coprocessor error traps**

■ Overview of Traps

Traps originate from within the instruction sequence. Traps are processed by first saving the necessary information to resume processing from the next instruction in the sequence, and then starting the processing routine corresponding to the type of trap that has occurred.

Branching to the exception processing routine takes place after execution of the instruction that has caused the exception.

The address of the instruction in which the exception occurs becomes the program counter (PC) value that is saved to the stack.

■ Sources of Traps

Sources of traps include the following:

- INT instructions (For details, see Section "4.5.1 "INT" Instructions")
- INTE instructions (For details, see Section "4.5.2 "INTE" Instruction")
- Step trace traps (For details, see Section "4.5.3 Step Trace Traps")
- Coprocessor not found traps (For details, see Section "4.5.4 Coprocessor Not Found Traps")
- Coprocessor error traps (For details, see Section "4.5.5 Coprocessor Error Trap")

4.5.1 "INT" Instructions

The "INT" instruction is used to create a software trap.
This section describes the operation, time requirements, program counter (PC) values saved, and other information of the "INT" instruction.

■ Overview of the "INT" Instruction

The "INT #u8" instruction is used to create a software trap with the interrupt number designated in the operand.

■ "INT" Instruction Operation

When the "INT #u8" instruction is executed, the following operations take place.

(1) The contents of the program status (PS) are saved to the system stack.

(2) The address of the next instruction is saved to the system stack.

(3) The value of the system stack pointer (SSP) is reduced by 8.

(4) The value "0" is written to the "I" flag in the condition code register (CCR) in the "PS".

(5) The value "0" is written to the "S" flag in the "CCR" in the "PS".

(6) The value "TBR + $3FC_H$ – 4 × u8" is stored in "PC".

■ Time to Start of Trap Processing for "INT" Instructions

The time required to start trap processing is 6 cycles.

■ "PC" Values Saved for "INT" Instruction Execution

The "PC" value saved to the system stack represents the address of the next instruction after the "INT" instruction.

■ Precautionary Information for Use of "INT" Instructions

The "INT" instruction should not be used within an "INTE" instruction handler or step trace trap-handler routine. This will prevent normal operation from resuming after the "RETI" instruction.

4.5.2 "INTE" Instruction

The "INTE" instruction is used to create a software trap for debugging.
This section describes the operation, time requirements, program counter (PC) values saved, and other information of the "INTE" instruction.

■ Overview of the "INTE" Instruction

The "INTE" instruction is used to create a software trap for debugging. This instruction allows the use of emulators.

This technique can be utilized by users for systems that have not been debugged by emulators.

■ "INTE" Instruction Operation

When the "INTE" instruction is executed, the following operations take place.

(1) The contents of the program status (PS) are saved to the system stack.
(2) The address of the next instruction is saved to the system stack.
(3) The value of the system stack pointer (SSP) is reduced by 8.
(4) The value "4" is written to the interrupt level mask register (ILM) in the "PS".
(5) The value "0" is written to the "S" flag in the "CCR" in the "PS".
(6) The value "TBR + $3D8_H$" is stored in "PC".

■ Time to Start of Trap Processing for "INTE" Instructions

The time required to start trap processing is 6 cycles.

■ "PC" Values Saved for "INTE" Instruction Execution

The "PC" value saved to the system stack represents the address of the next instruction after the "INTE" instruction.

■ Precautionary Information for Use of "INTE" Instructions

The "INTE" instruction cannot be used in user programs involving debugging with an emulator. Also, the "INTE" instruction should not be used within an "INTE" instruction handler or step trace trap-handler routine. This will prevent normal operation from resuming after the "RETI" instruction. Note also that no "EIT" events can be generated by "INTE" instructions during stepwise execution.

4.5.3 Step Trace Traps

Step trace traps are traps used by debuggers. This type of trap can be created for each individual instruction in a sequence by setting the "T" flag in the system condition code register (SCR) in the program status (PS).
This section describes conditions for the generation, operations, program counter (PC) values saved, and other information of step trace traps.

■ Overview of Step Trace Traps

Step trace traps are traps used by debuggers. This type of trap can be created for each individual instruction in a sequence, by setting the "T" flag in the "SCR" in the "PS".

In the execution of delayed branching instructions, step trace traps are not generated immediately after the execution of branching. The trap is generated after execution of the instruction(s) in the delay slot.

The step trace trap can be utilized by users for systems that have not been debugged by emulators.

■ Conditions for Generation of Step Trace Traps

A step trace trap is generated when the following conditions are met.

- The "T" flag in the "SCR" in the "PS" is set to "1".
- The currently executing instruction is not a delayed branching instruction.
- The CPU is not processing an "INTE" instruction or a step trace trap processing routine.

■ Step Trace Trap Operation

When a step trace trap is generated, the following operations take place.

(1) The contents of the program status (PS) are saved to the system stack.
(2) The address of the next instruction is saved to the system stack.
(3) The value of the system stack pointer (SSP) is reduced by 8.
(4) The value "0" is written to the "S" flag in the "CCR" in the "PS".
(5) The value "TBR + $3C4_H$" is stored in "PC".

■ "PC" Values Saved for Step Trace Traps

The "PC" value saved to the system stack represents the address of the next instruction after the step trace trap.

■ Relation of Step Trace Traps to "NMI" and External Interrupts

When the "T" flag is set to enable step trace traps, both "NMI" and external interrupts are disabled.

■ Precautionary Information for Use of Step Trace Traps

Step trace traps cannot be used in user programs involving debugging with an emulator. Note also that no "EIT" events can be generated by "INTE" instructions when the step trace trap function is used.

4.5.4 Coprocessor Not Found Traps

Coprocessor not found traps are generated by executing coprocessor instructions using coprocessors not found in the system.
This section describes conditions for the generation of coprocessor not found traps, in addition to operation, program counter (PC) values saved, and other information.

■ Overview of Coprocessor Not Found Traps

Coprocessor not found traps are generated by executing coprocessor instructions using coprocessors not found in the system.

■ Conditions for Generation of Coprocessor Not Found Traps

A coprocessor not found trap is generated when the following conditions are met.

- Execution of a "COPOP/COPLD/COPST/COPSV" instruction.
- No coprocessor present in the system corresponds to the operand "#u4" in any of the above instructions.

■ Coprocessor Not Found Trap Operation

When a coprocessor not found trap is generated, the following operations take place.

(1) The contents of the program status (PS) are saved to the system stack.
(2) The address of the next instruction is saved to the system stack.
(3) The value of the system stack pointer (SSP) is reduced by 8.
(4) The value "0" is written to the "S" flag in the condition code register (CCR) in the "PS".
(5) The value "TBR + $3E0_H$" is stored in "PC".

■ "PC" Values Saved for Coprocessor Not Present Traps

The "PC" value saved to the system stack represents the address of the next instruction after the coprocessor instruction that caused the trap.

■ General-purpose Registers during Execution of "COPST/COPSV" Instructions

Execution of any "COPST/COPSV" instruction referring to a coprocessor that is not present in the system will cause undefined values to be transferred to the general-purpose register (R0 to R14) designated in the operand. The coprocessor not found trap will be activated after the designated general-purpose register is updated.

4.5.5 Coprocessor Error Trap

A coprocessor error trap is generated when an error has occurred in a coprocessor operation and the CPU executes another coprocessor instruction involving the same coprocessor.
This section describes conditions for the generation, operations, and program counter (PC) values saved of coprocessor error traps.

■ Overview of Coprocessor Error Traps

A coprocessor error trap is generated when an error has occurred in a coprocessor operation and the CPU executes another coprocessor instruction involving the same coprocessor. Note that no coprocessor error traps are generated for execution of "COPSV" instructions.

■ Conditions for Generation of Coprocessor Error Traps

A coprocessor error trap is generated when the following conditions are met.

- An error has occurred in coprocessor operation.
- A "COPOP/COPLD/COPST" instruction is executed involving the same coprocessor.

■ Coprocessor Error Trap Operation

When a coprocessor error trap is generated, the following operations take place.

(1) The contents of the program status (PS) are saved to the system stack.
(2) The address of the next instruction is saved to the system stack.
(3) The value of the system stack pointer (SSP) is reduced by 8.
(4) The value "0" is written to the "S" flag in the condition code register (CCR) in the "PS".
(5) The value "TBR + $3DC_H$" is stored in "PC".

■ "PC" Values Saved for Coprocessor Error Traps

The "PC" value saved to the system stack represents the address of the next instruction after the coprocessor instruction that caused the trap.

■ Results of Coprocessor Operations after a Coprocessor Error Trap

Despite the occurrence of a coprocessor error trap, the execution of the coprocessor instruction ("COPOP/COPLD/COPST") remains valid and the results of the instruction are retained. Note that the results of operations affected by the coprocessor error will not be correct.

■ Saving and Restoring Coprocessor Error Information

When a coprocessor is used in a multi-tasking environment, the internal resources of the coprocessor become part of the system context. Thus whenever context switching occurs, it is necessary to save or restore the contents of the coprocessor. Problems arise when there are hidden coprocessor errors remaining from former tasks at the time of context switching.

In such cases, when the exception is detected in a coprocessor context save instruction by the dispatcher, it becomes impossible to return the information to the former task. This problem is avoided by executing a "COPSV" instruction, which does not send notification of coprocessor errors but acts to clear the internal error. Note that the error information is retained in the status information that is saved. If the saved status information is returned to the coprocessor at the time of re-dispatching to the former task, the hidden error condition is cleared and the CPU is notified when the next coprocessor instruction is executed.

Figure 4.5-1 shows an example in which notification to the coprocessor does not succeed, and Figure 4.5-2 illustrates the use of the "COPSV" instruction to save and restore error information.

Figure 4.5-1 Example: Coprocessor Error Notification Not Successful

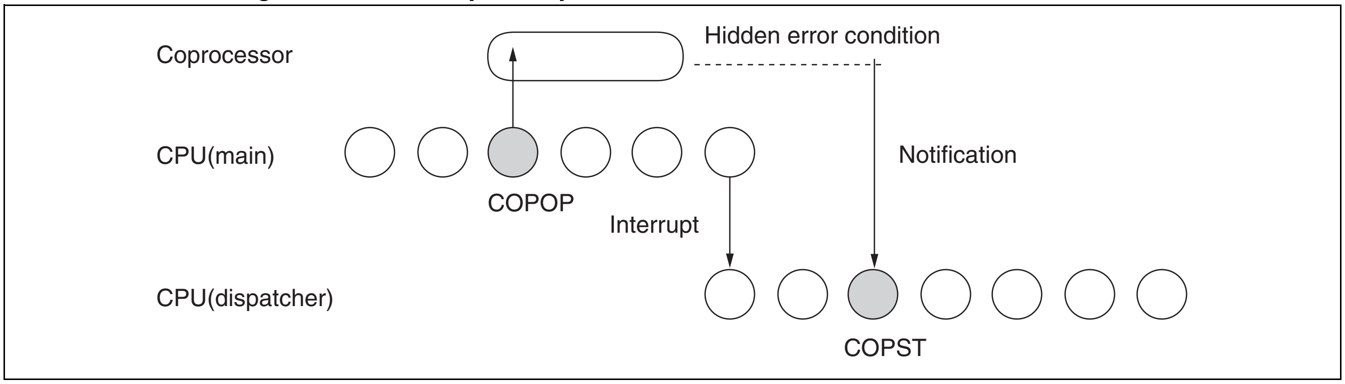
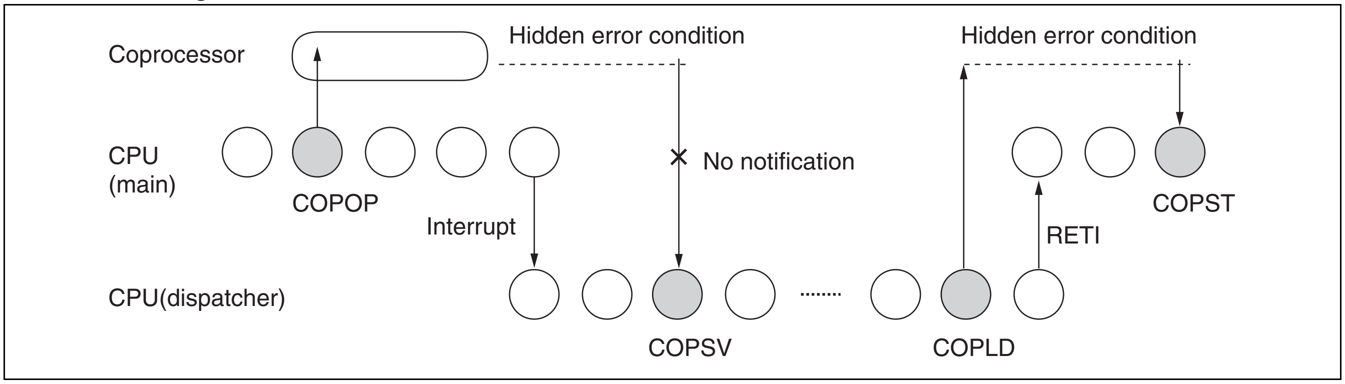


Figure 4.5-2 Use of "COPSV" Instruction to Save and Restore Error Information



4.6 Priority Levels

When multiple "EIT" requests occur at the same time, priority levels are used to select one source and execute the corresponding "EIT" sequence. After the "EIT" sequence is completed, "EIT" request detection is applied again to enable processing of multiple "EIT" requests.
Acceptance of certain types of "EIT" requests can mask other factors. In such cases the priority applied by the "EIT" processing handler may not match the priority of the requests.

■ Priority of Simultaneous Occurrences

The FR family uses a hardware function to determine the priority of acceptance of "EIT" requests.

Table 4.6-1 shows the priority levels of "EIT" requests.

Table 4.6-1 Priority of "EIT" Requests

Priority	Source	Masking of other sources
1	Reset	Other sources discarded
2	Undefined instruction exception	Other sources disabled
3	INT instruction	I flag = 0
	Coprocessor not found trap Coprocessor error trap	None
4	User interrupt	ILM = level of source accepted
5	NMI	ILM = 15
6	Step trace trap	ILM = 4
7	INTE instruction	ILM = 4

■ Priority of Multiple Processes

When the acceptance of an "EIT" source results in the masking of other sources, the priority of execution of simultaneously occurring "EIT" handlers is as shown in Table 4.6-2.

Table 4.6-2 Priority of Execution of "EIT" Handlers

Priority	Source	Masking of other sources
1	Reset	Other sources discarded
2	Undefined instruction exception	Other sources disabled
3	Step trace trap	ILM = 4 *
4	INTE instruction	ILM = 4 *
5	NMI	ILM = 15
6	INT instruction	I flag = 0
7	User interrupt	ILM = level of source accepted
8	Coprocessor not found trap Coprocessor error trap	None

*: When "INTE" instructions are run stepwise, only the step trace "EIT" is generated.
Sources related to the "INTE" instruction will be ignored.

CHAPTER 5
PRECAUTIONARY INFORMATION FOR THE FR FAMILY CPU

This chapter presents precautionary information related to the use of the FR family CPU.

5.1 Pipeline Operation

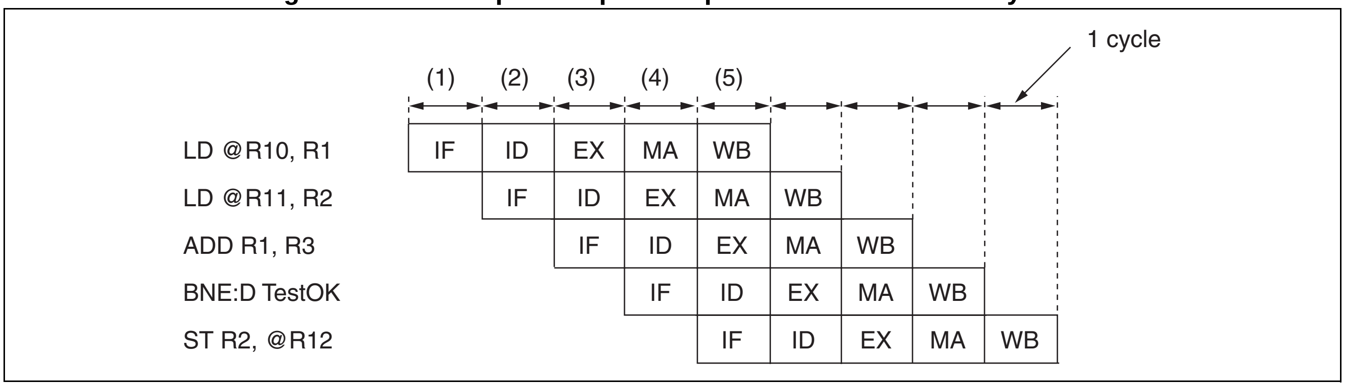
The FR family CPU processes all instructions using a 5-stage pipeline operation. This makes it possible to process nearly all instructions within one cycle.

■ Overview of Pipeline Operation

In a pipeline operation the steps by which the CPU interprets and executes instructions are divided into several cycles, so that instructions can be processed simultaneously in successive cycles. This enables the system to appear to execute in one cycle many instructions that would require several cycles in other methods of processing. The FR family CPU simultaneously executes five types (IF, ID, EX, MA, and WB) of processing cycles, as shown in Figure 5.1-1. This is referred to as five-stage pipeline processing.

- IF: Load instruction
- ID: Interpret instruction
- EX: Execute instruction
- MA: Memory access
- WB: Write to register

Figure 5.1-1 Example of Pipeline Operation in the FR Family CPU



● Processes occurring in each 1 cycle in the above example:

(1) Load instruction "LD @R10,R1"

(2) Interpret instruction "LD @R10,R1" Load instruction "LD, @R11,R2"

(3) Execute instruction "LD @R10,R1" Interpret instruction "LD, @R11,R2"
Load instruction, "ADD R1, R3"

(4) Memory access instruction "LD @R10,R1" Execute instruction "LD, @R11,R2"
Interpret instruction, "ADD R1, R3" Load instruction "BNE:D TestOK"

(5) Write instruction "LD @R10,R1" to register Memory access instruction "LD, @R11,R2"
Execute instruction, "ADD R1, R3" Interpret instruction, "BNE:D TestOK"
Load instruction "ST R2,@R12"

5.2 Pipeline Operation and Interrupt Processing

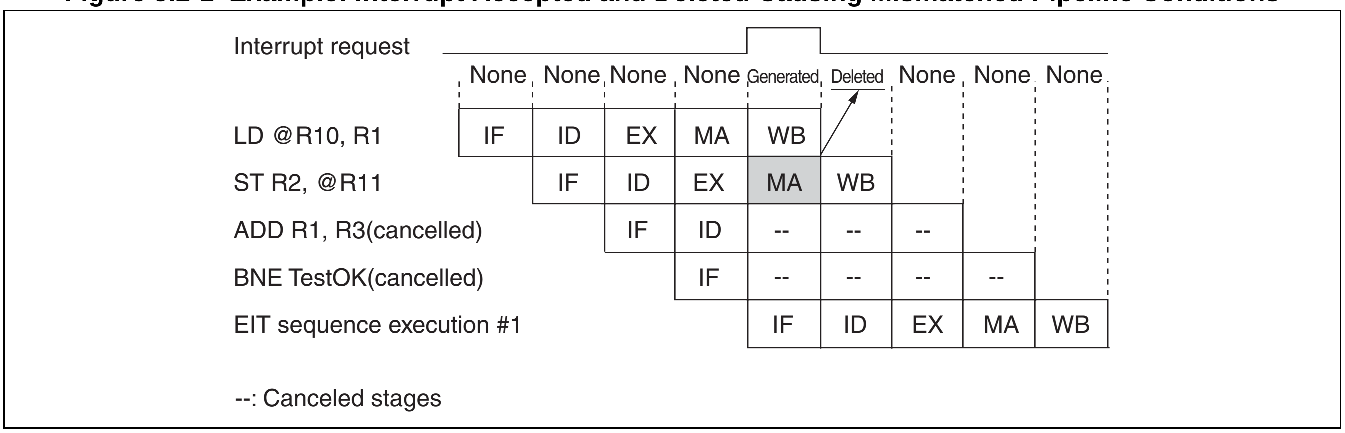
The FR family CPU processes all instructions through pipeline operation. Therefore, particularly for instructions that start hardware events, it is possible for contradictory conditions to exist before and after an instruction.

■ Precautionary Information for Interrupt Processing in Pipeline Operation

Because the FR family CPU operates in pipeline mode, the recognition of an interrupt signal is preceded by several instructions in respective states of pipeline processing. If one of those instructions being executed in the pipeline acts to delete the interrupt, the CPU will branch normally to the respective interrupt processing program but when control is transferred to interrupt processing the interrupt request will no longer be effective.

Note that this type of condition does not occur in exception or trap processing.

Figure 5.2-1 Example: Interrupt Accepted and Deleted Causing Mismatched Pipeline Conditions



■ Conditions that Are Actually Generated

The following processing conditions may cause an interrupt to be deleted after acceptance.

- A program that clears interrupt sources while in interrupt-enabled mode
- Writing to an interrupt-enable bit in a peripheral function while in interrupt-enabled mode

■ How to Avoid Mismatched Pipeline Conditions

To avoid deleting interrupts that have already been accepted, programmers should use the "I" flag in the condition code register (CCR) in the program status (PS) to regulate interrupt sources.

5.3 Register Hazards

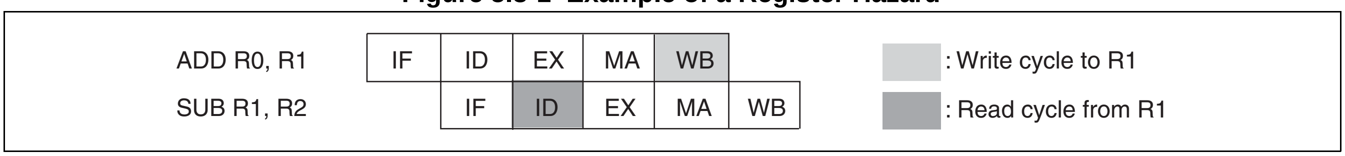
The FR family CPU executes program steps in the order in which they are written, and is therefore equipped with a function that detects the occurrence of register hazards and stops pipeline processing when necessary. This enables programs to be written without attention to the order in which registers are used

■ Overview of Register Hazards

The CPU in pipeline operation may simultaneously process one instruction that involves writing values to a register, and a subsequent instruction that attempts to refer to the same register before the write process is completed. This is called a register hazard.

In the example in Figure 5.3-1, the program will read the address value at "R1" before the desired value has been written to "R1" by the previous instruction. As a result, the old value at "R1" will be read instead of the new value.

Figure 5.3-1 Example of a Register Hazard

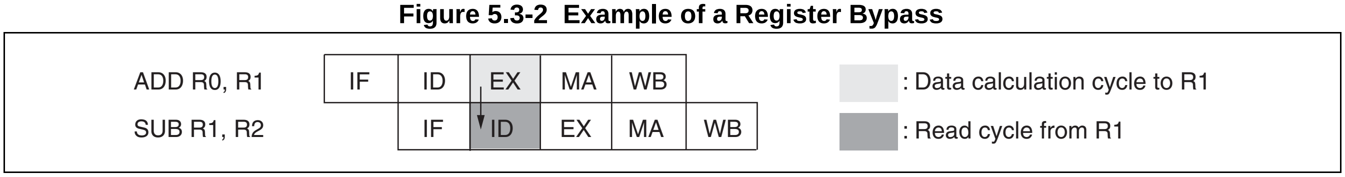


■ Register Bypassing

Even when a register hazard does occur, it is possible to process instructions without operating delays if the data intended for the register to be accessed can be extricated from the preceding instruction. This type of data transfer processing is called register bypassing, and the FR family CPU is equipped with a register bypass function.

In the example in Figure 5.3-2, instead of reading the "R1" in the "ID" stage of the "SUB" instruction, the program uses the results of the calculation from the "EX" stage of the "ADD" instruction (before the results are written to the register) and thus executes the instruction without delay.

Figure 5.3-2 Example of a Register Bypass

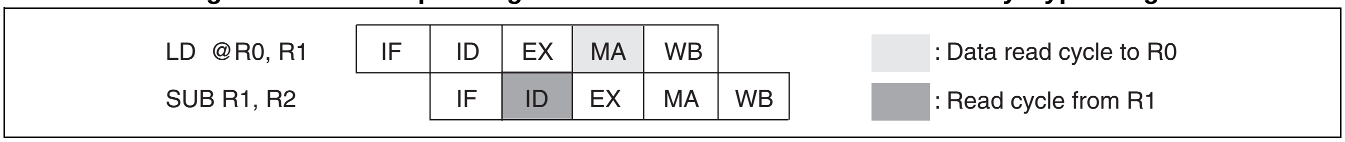


■ Interlocking

Instructions which are relatively slow in loading data to the CPU may cause register hazards that cannot be handled by register bypassing.

In the example in Figure 5.3-3, data required for the "ID" stage of the "SUB" instruction must be loaded to the CPU in the "MA" stage of the "LD" instruction, creating a hazard that cannot be avoided by the bypass function.

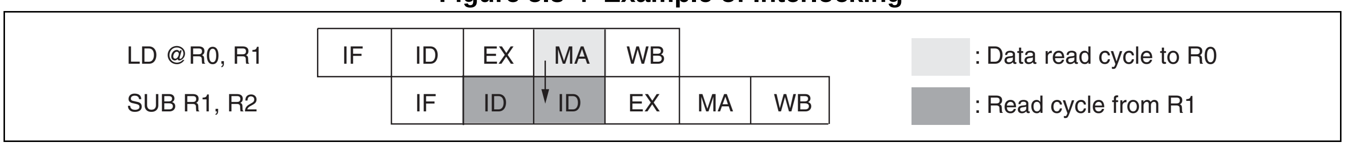
Figure 5.3-3 Example: Register Hazard that Cannot be Avoided by Bypassing



In cases such as this, the FR family CPU executes the instruction correctly by pausing before execution of the subsequent instruction. This function is called interlocking.

In the example in Figure 5.3-4, the "ID" stage of the "SUB" instruction is delayed until the data is loaded from the "MA" stage of the "LD" instruction.

Figure 5.3-4 Example of Interlocking



■ Interlocking Produced by Reference to "R15" and General-purpose Registers after Changing the "S" Flag

The general-purpose register "R15" is designed to function as either the system stack pointer (SSP) or user stack pointer (USP). For this reason, the FR family CPU is designed to automatically generate an interlock whenever a change to the "S" flag in the condition code register (CCR) in the program status (PS) is followed immediately by an instruction that references the "R15". This interlock enables the CPU to reference the "SSP" or "USP" values in the order in which they are written in the program. FR family hardware design similarly generates an interlock whenever a TYPE-A format instruction immediately follows an instruction that changes the value of the "S" flag.

For information on instruction format types, see Section "6.1 Instruction Formats".

5.4 Delayed Branching Processing

Because the FR family CPU features pipeline operation, branching instructions must first be loaded before they are executed. Delayed branching processing is the function to execute the loaded instruction, and allows to accelerate processing speeds.

■ Overview of Branching with Non-delayed Branching Instructions

In a pipeline operation, by the time the CPU recognizes an instruction as a branching instruction the next instruction has already been loaded. To process the program as written, the instruction following the branching instruction must be canceled in the middle of execution. Branching instructions that are handled in this manner are non-delayed branching instructions.

Examples of processing non-delayed branching instructions (both when branching conditions are satisfied and not satisfied) are described in Section "5.4.1 Processing Non-delayed Branching Instructions".

■ Overview of Branching with Delayed Branching Instructions

An instruction immediately following a branching instruction will already be loaded by the CPU by the time the branching instruction is executed. This position is called the delay slot.

A delayed branching instruction is a branching instruction that executes the instruction in the delay slot regardless of whether the branching conditions are satisfied or not satisfied.

Examples of processing delayed branching instructions (both when branching conditions are satisfied and not satisfied) are described in Section "5.4.2 Processing Delayed Branching Instructions".

■ Instructions Prohibited in Delay Slots

The following instructions may not be used in delayed branching processing by the FR family CPU.

- LDI:32 #i32, Ri LDI:20 #i20, Ri
- COPOP #u4, #CC, CRj, CRi
  COPLD #u4, #CC, Rj, CRi
  COPST #u4, #CC, CRj, Ri
  COPSV #u4, #CC, CRj, Ri
- JMP @Ri
  CALL label12
  CALL @Ri
  RET
  Conditional branching instruction and related delayed branching instructions
- INT #u8
  RETI
  INTE

- AND Rj, @Ri
  ANDH Rj, @Ri
  ANDB Rj, @Ri
  OR Rj, @Ri
  ORH Rj, @Ri
  ORB Rj, @Ri
  EOR Rj, @Ri
  EORH Rj, @Ri
  EORB Rj, @Ri
- BANDH #u4, @Ri
  BANDL #u4, @Ri
  BORH #u4, @Ri
  BORL #u4, @Ri
  BEORH #u4, @Ri
  BEORL #u4, @Ri
  BTSTH #u4, @Ri
  BTSTL #u4, @Ri
- MUL Rj, Ri
  MULU Rj, Ri
  MULH Rj, Ri
  MULUH Rj, Ri
- LD @R15+, PS
- LDM0 (reglist)
  LDM1 (reglist)
  STM0 (reglist)
  STM1 (reglist)
  ENTER #u10
  XCHB @Rj, Ri
- DMOV @dir10, @R13+
  DMOV @R13+, @dir10
  DMOV @dir10, @-R15
  DMOV @R15+, @dir10
  DMOVH @dir9, @R13+
  DMOVH @R13+, @dir9
  DMOVB @dir8, @R13+
  DMOVB @R13+, @dir8

■ Restrictions on Interrupts during Processing of Delayed Branching Instructions

"EIT" processing is not accepted during execution of delayed branching instructions or delayed branching processing.

5.4.1 Processing Non-delayed Branching Instructions

The FR family CPU processes non-delayed branching instructions in the order in which the program is written, introducing a 1-cycle delay in execution speed if branching takes place.

■ Examples of Processing Non-delayed Branching Instructions

Figure 5.4-1 shows an example of processing a non-delayed branching instruction when branching conditions are satisfied.

In this example, the instruction "ST R2,@R12" (which immediately follows the branching instruction) has entered the pipeline operation before the fetching of the branch destination instruction, but is canceled during execution.

As a result, the program is processed in the order in which it is written, and the branching instruction requires an apparent processing time of two cycles.

Figure 5.4-1 Example: Processing a Non-delayed Branching Instruction (Branching Conditions Satisfied)

LD @R10, R1	IF	ID	EX	MA	WB					
LD @R11, R2		IF	ID	EX	MA	WB				
ADD R1, R3			IF	ID	EX	MA	WB			
BNE TestOK(branching conditions satisfied)				IF	ID	EX	MA	WB		
ST R2, @R12(instruction immediately after)					IF	--	--	--	--	
ST R2, @R13(branch destination instruction)						IF	ID	EX	MA	WB

-- : Canceled stages

: PC change

Figure 5.4-2 shows an example of processing a non-delayed branching instruction when branching conditions are not satisfied.

In this example, the instruction "ST R2,@R12" (which immediately follows the branching instruction) has entered the pipeline operation before the fetching of the branch destination instruction, and is executed without being canceled.

Because instructions are executed without branching, the program is processed in the order in which it is written. The branching instruction requires an apparent processing time of one cycle.

Figure 5.4-2 Example: Processing a Non-delayed Branching Instruction (Branching Conditions Not Satisfied)

LD @R10, R1	IF	ID	EX	MA	WB						
LD @R11, R2		IF	ID	EX	MA	WB					
ADD R1, R3			IF	ID	EX	MA	WB				
BNE TestOK(branching conditions not satisfied)				IF	ID	EX	MA	WB			
ST R2, @R12(instruction immediately after)					IF	ID	EX	MA	WB		Not canceled
ADD #4, R12(subsequent instruction)						IF	ID	EX	MA	WB	

5.4.2 Processing Delayed Branching Instructions

The FR family CPU processes delayed branching instructions with an apparent execution speed of 1 cycle, regardless of whether branching conditions are satisfied or not satisfied. When branching occurs, this is one cycle faster than using non-delayed branching instructions.
However, the apparent order of instruction processing is inverted in cases where branching occurs.

■ Examples of Processing Delayed Branching Instructions

Figure 5.4-3 shows an example of processing a delayed branching instruction when branching conditions are satisfied.

In this example, the branch destination instruction, "ST R2,@R13" is executed after the instruction "ST R2,@R12" in the delay slot. As a result, the branching instruction has an apparent execution speed of one cycle. However, the instruction "ST R2,@R12" in the delay slot is executed before the branch destination instruction "ST R2,@R13" and therefore the apparent order of processing is inverted.

Figure 5.4-3 Example: Processing a Delayed Branching Instruction (Branching Condition Satisfied)

LD @R10, R1	IF	ID	EX	MA	WB						
LD @R11, R2		IF	ID	EX	MA	WB					
ADD R1, R3			IF	ID	EX	MA	WB				
BNE:D TestOK(branching conditions satisfied)				IF	ID	EX	MA	WB			
ST R2, @R12(delay slot instruction)					IF	ID	EX	MA	WB		Not canceled
ST R2, @R13(branch destination instruction)						IF	ID	EX	MA	WB	

(shaded) : PC change

Figure 5.4-4 shows an example of processing a delayed branching instruction when branching conditions are not satisfied.

In this example the delay slot instruction "ST R2,@R12" is executed without being canceled. As a result, the program is processed in the order in which it is written. The branching instruction requires an apparent processing time of one cycle.

Figure 5.4-4 Example: Processing a Delayed Branching Instruction (Branching Conditions Not Satisfied)

LD @R10, R1	IF	ID	EX	MA	WB						
LD @R11, R2		IF	ID	EX	MA	WB					
ADD R1, R3			IF	ID	EX	MA	WB				
BNE:D TestOK (branching conditions not satisfied)				IF	ID	EX	MA	WB			
ST R2, @R12 (delay slot instruction)					IF	ID	EX	MA	WB		Not canceled
ADD #4, R12						IF	ID	EX	MA	WB	

■ Examples of Programing Delayed Branching Instructions

An example of programing a delayed branching instruction is shown below.

```
          .
          .
LD        @R10, R1
LD        @R11, R2
ADD       R1, R3
BNE:D     TestOK
ST        R2, @R12
ADD       #4, R12      ; not satisfy
          .
          .
          .
TestOK:                ; satisfied
ST        R2, @R13
          .
          .
```

CHAPTER 6 INSTRUCTION OVERVIEW

This chapter presents an overview of the instructions used with the FR family CPU.
All FR family CPU instructions are in 16-bit fixed length format, except for immediate data transfer instructions which may exceed 16 bits in length. This format enables the creation of a compact object code and smoother pipeline processing.

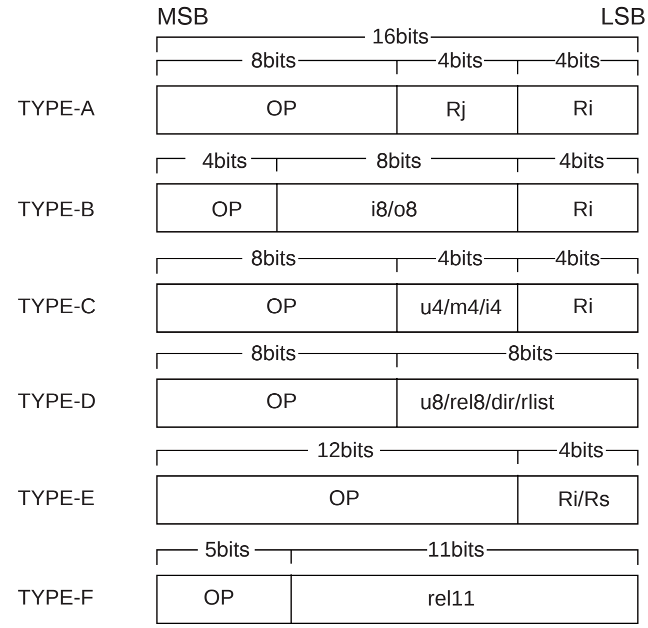
6.1 Instruction Formats

The FR family CPU uses six types of instruction format, TYPE-A through TYPE-F.

■ Instruction Formats

All instructions used by the FR family CPU are written in the six formats shown in Figure 6.1-1.

Figure 6.1-1 Instruction Formats



■ Relation between Bit Patterns "Ri" and "Rj" and Register Values

Table 6.1-1 shows the relation between general-purpose register numbers and field bit pattern values.

Table 6.1-1 General-purpose Register Numbers and Field Bit Pattern Values

Ri/Rj	Register	Ri/Rj	Register	Ri/Rj	Register	Ri/Rj	Register
0000	R0	0100	R4	1000	R8	1100	R12
0001	R1	0101	R5	1001	R9	1101	R13
0010	R2	0110	R6	1010	R10	1110	R14
0011	R3	0111	R7	1011	R11	1111	R15

■ Relation between Bit Pattern "Rs" and Register Values

Table 6.1-2 shows the relation between dedicated register numbers and field bit pattern values.

Table 6.1-2 Dedicated Register Numbers and Field Bit Pattern Values

Rs	Register	Rs	Register	Rs	Register	Rs	Register
0000	TBR	0100	MDH	1000	reserved	1100	reserved
0001	RP	0101	MDL	1001	reserved	1101	reserved
0010	SSP	0110	reserved	1010	reserved	1110	reserved
0011	USP	0111	reserved	1011	reserved	1111	reserved

Note: Bit patterns marked "reserved" are reserved for system use. Proper operation is not assured if these patterns are used in programming.

6.2 Instruction Notation Formats

FR family CPU instructions are written in the following three notation formats.

- **Calculations are designated by a mnemonic placed between operand 1 and operand 2, with the results stored at operand 2.**
- **Operations are designated by a mnemonic, and use operand 1.**
- **Operations are designated by a mnemonic.**

■ Instruction Notation Formats

FR family CPU instructions are written in the following 3 notation formats.

● Calculations are designated by a mnemonic placed between operand 1 and operand 2, with the results stored at operand 2.

	<Mnemonic>	<Operand 1>	<Operand 2>	
[Example]	ADD	R1,	R2	; R1 + R2 --> R2

● Operations are designated by a mnemonic, and use operand 1.

	<Mnemonic>	<Operand 1>	
[Example]	JMP	@R1	; R1 --> PC

● Operations are designated by a mnemonic.

	<Mnemonic>	
[Example]	NOP	; No operation

CHAPTER 7

DETAILED EXECUTION INSTRUCTIONS

This chapter presents each of the execution instructions used by the FR family assembler, in reference format.
The execution instructions used by the FR family CPU are classified as follows.

- **Add/Subtract Instructions**
- **Compare Instructions**
- **Logical Calculation Instructions**
- **Bit Operation Instructions**
- **Multiply/Divide Instructions**
- **Shift Instructions**
- **Immediate Data Transfer Instructions**
- **Memory Load Instructions**
- **Memory Store Instructions**
- **Inter-register Transfer Instructions/Dedicated Register Transfer Instructions**
- **Non-delayed Branching Instructions**
- **Delayed Branching Instructions**
- **Direct Addressing Instructions**
- **Resource Instructions**
- **Coprocessor Instructions**
- **Other Instructions**

7.1 ADD (Add Word Data of Source Register to Destination Register)

Adds word data in "Rj" to word data in "Ri", stores results to "Ri".

■ ADD (Add Word Data of Source Register to Destination Register)

Assembler format: ADD Rj, Ri

Operation: Ri + Rj → Ri

Flag change:

N	Z	V	C
C	C	C	C

N : Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z : Set when the operation result is "0", cleared otherwise.

V : Set when an overflow has occurred as a result of the operation, cleared otherwise.

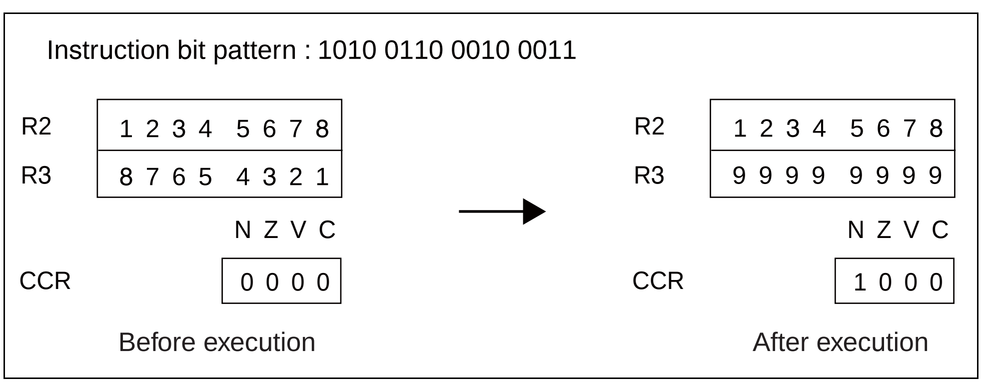
C : Set when a carry has occurred as a result of the operation, cleared otherwise.

Execution cycles: 1 cycle

Instruction format:

MSB									LSB
1	0	1	0	0	1	1	0	Rj	Ri

Example: ADD R2, R3



7.2 ADD (Add 4-bit Immediate Data to Destination Register)

Adds the result of the higher 28 bits of 4-bit immediate data with zero extension to the word data in "Ri", stores results to the "Ri".

■ ADD (Add 4-bit Immediate Data to Destination Register)

Assembler format: ADD #i4, Ri

Operation: Ri + extu(i4) → Ri

Flag change:

N	Z	V	C
C	C	C	C

N : Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z : Set when the operation result is "0", cleared otherwise.

V : Set when an overflow has occurred as a result of the operation, cleared otherwise.

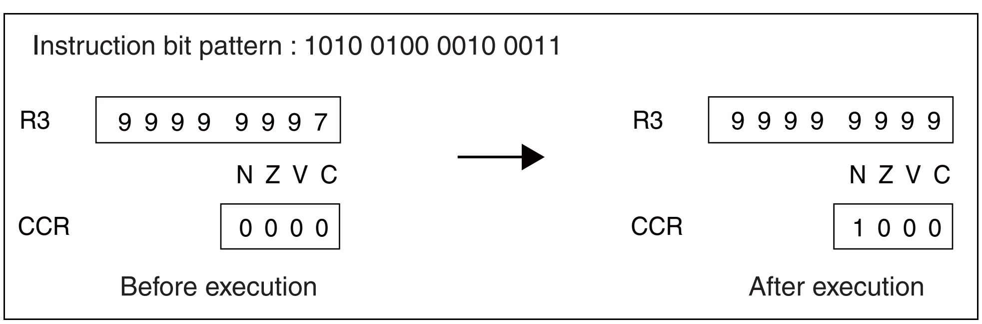
C : Set when a carry has occurred as a result of the operation, cleared otherwise.

Execution cycles: 1 cycle

Instruction format:

MSB									LSB
1	0	1	0	0	1	0	0	i4	Ri

Example: ADD #2, R3



7.3 ADD2 (Add 4-bit Immediate Data to Destination Register)

Adds the result of the higher 28 bits of 4-bit immediate data with minus extension to the word data in "Ri", stores results to "Ri".
The way a "C" flag of this instruction varies is the same as the ADD instruction ; it is different from that of the SUB instruction.

■ ADD2 (Add 4-bit Immediate Data to Destination Register)

Assembler format: ADD2 #i4, Ri

Operation: Ri + extn(i4) → Ri

Flag change:

N	Z	V	C
C	C	C	C

N : Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z : Set when the operation result is "0", cleared otherwise.

V : Set when an overflow has occurred as a result of the operation, cleared otherwise.

C : Set when a carry has occurred as a result of the operation, cleared otherwise.

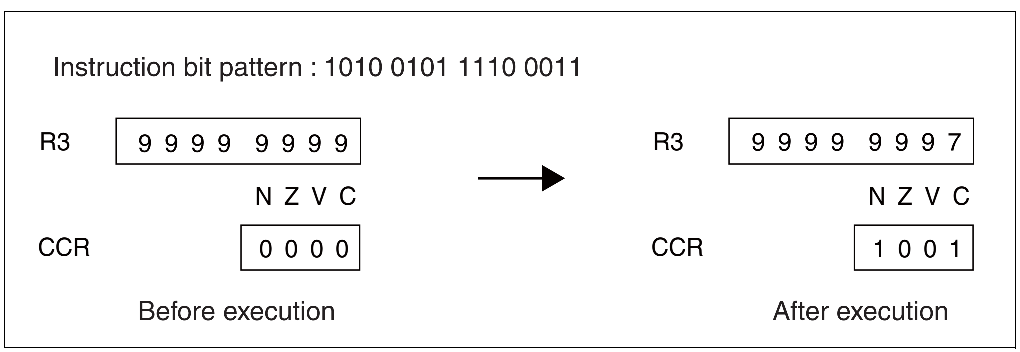
Execution cycles: 1 cycle

Instruction format:

MSB ... LSB

1	0	1	0	0	1	0	1	i4	Ri

Example: ADD2 #–2, R3



7.4 ADDC (Add Word Data of Source Register and Carry Bit to Destination Register)

Adds the word data in "Rj" to the word data in "Ri" and carry bit, stores results to "Ri".

■ ADDC (Add Word Data of Source Register and Carry Bit to Destination Register)

Assembler format: ADDC Rj, Ri

Operation: Ri + Rj + C → Ri

Flag change:

N	Z	V	C
C	C	C	C

N : Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z : Set when the operation result is "0", cleared otherwise.

V : Set when an overflow has occurred as a result of the operation, cleared otherwise.

C : Set when a carry has occurred as a result of the operation, cleared otherwise.

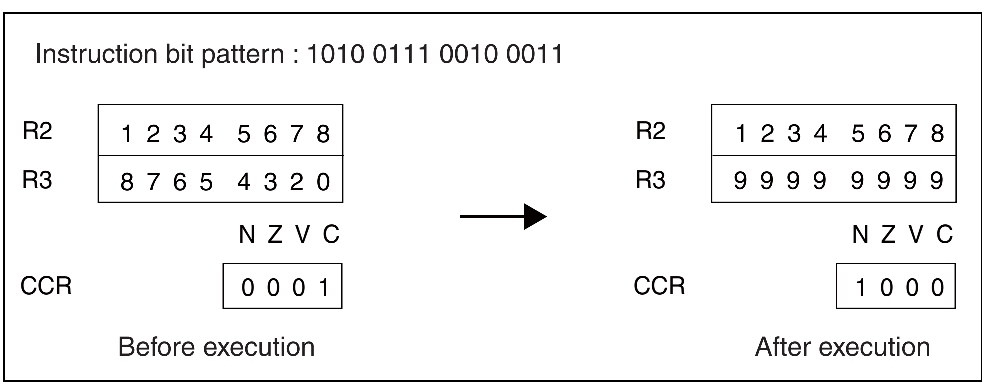
Execution cycles: 1 cycle

Instruction format:

MSB ... LSB

1	0	1	0	0	1	1	1	Rj	Ri

Example: ADDC R2, R3



7.5 ADDN (Add Word Data of Source Register to Destination Register)

Adds the word data in "Rj" and the word data in "Ri", stores results to "Ri" without changing flag settings.

■ ADDN (Add Word Data of Source Register to Destination Register)

Assembler format: ADDN Rj, Ri

Operation: Ri + Rj → Ri

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

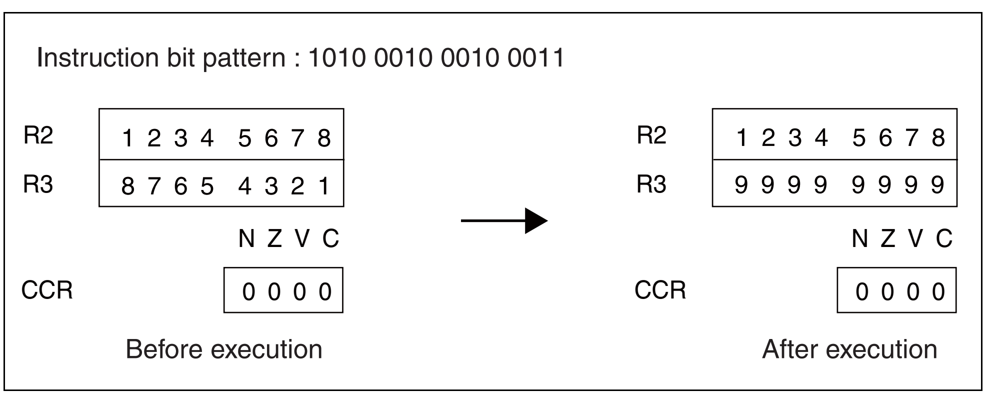
Execution cycles: 1 cycle

Instruction format:

MSB ... LSB

1	0	1	0	0	0	1	0	Rj	Ri

Example: ADDN R2, R3



7.6 ADDN (Add Immediate Data to Destination Register)

Adds the result of the higher 28 bits of 4-bit immediate data with zero extension to the word data in "Ri", stores the results to "Ri" without changing flag settings.

■ ADDN (Add Immediate Data to Destination Register)

Assembler format: ADDN #i4, Ri

Operation: Ri + extu(i4) → Ri

Flag change:

N	Z	V	C
–	–	–	–

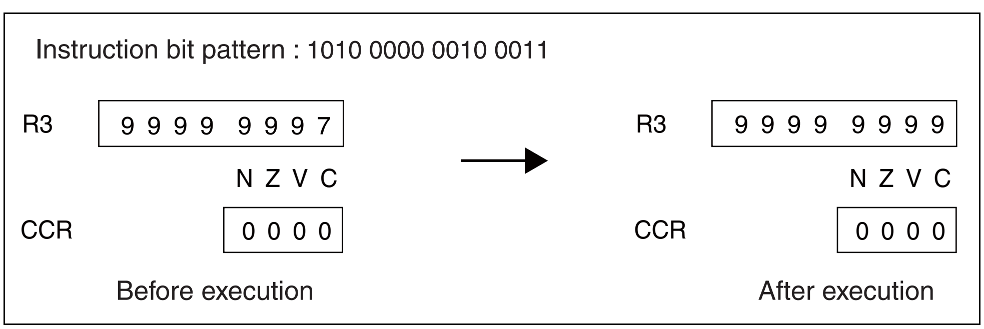
N, Z, V, and C: Unchanged

Execution cycles: 1 cycle

Instruction format:

MSB									LSB
1	0	1	0	0	0	0	0	i4	Ri

Example: ADDN #2, R3



7.7 ADDN2 (Add Immediate Data to Destination Register)

Adds the result of the higher 28 bits of 4-bit immediate data with minus extension to word data in "Ri", stores the results to "Ri" without changing flag settings.

■ ADDN2 (Add Immediate Data to Destination Register)

Assembler format: ADDN2 #i4, Ri

Operation: Ri + extn(i4) + → Ri

Flag change:

N	Z	V	C
–	–	–	–

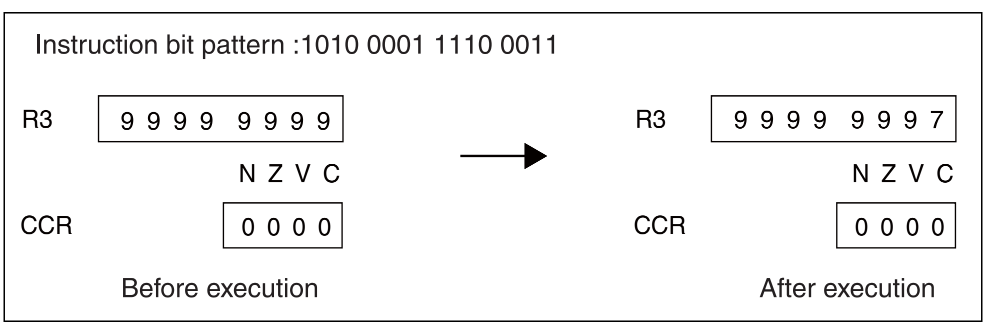
N, Z, V, and C: Unchanged

Execution cycles: 1 cycle

Instruction format:

MSB									LSB
1	0	1	0	0	0	0	1	i4	Ri

Example: ADDN2 #–2, R3



7.8 SUB (Subtract Word Data in Source Register from Destination Register)

Subtracts the word data in "Rj" from the word data in "Ri", stores results to "Ri".

■ SUB (Subtract Word Data in Source Register from Destination Register)

Assembler format: SUB Rj, Ri

Operation: Ri – Rj → Ri

Flag change:

N	Z	V	C
C	C	C	C

N : Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z : Set when the operation result is "0", cleared otherwise.

V : Set when an overflow has occurred as a result of the operation, cleared otherwise.

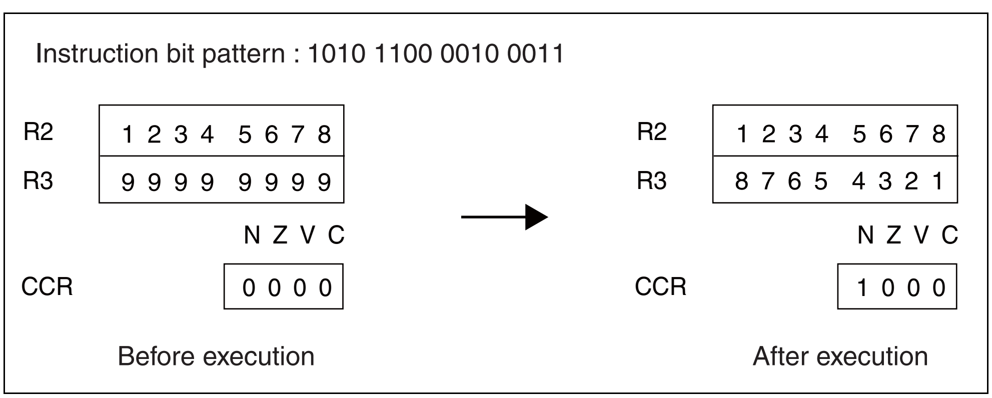
C : Set when a borrow has occurred as a result of the operation, cleared otherwise.

Execution cycles: 1 cycle

Instruction format:

MSB									LSB
1	0	1	0	1	1	0	0	Rj	Ri

Example: SUB R2, R3



7.9 SUBC (Subtract Word Data in Source Register and Carry Bit from Destination Register)

Subtracts the word data in "Rj" and the carry bit from the word data in "Ri", stores results to "Ri".

■ SUBC (Subtract Word Data in Source Register and Carry Bit from Destination Register)

Assembler format: SUBC Rj, Ri

Operation: Ri – Rj – C → Ri

Flag change:

N	Z	V	C
C	C	C	C

N : Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z : Set when the operation result is "0", cleared otherwise.

V : Set when an overflow has occurred as a result of the operation, cleared otherwise.

C : Set when a borrow has occurred as a result of the operation, cleared otherwise.

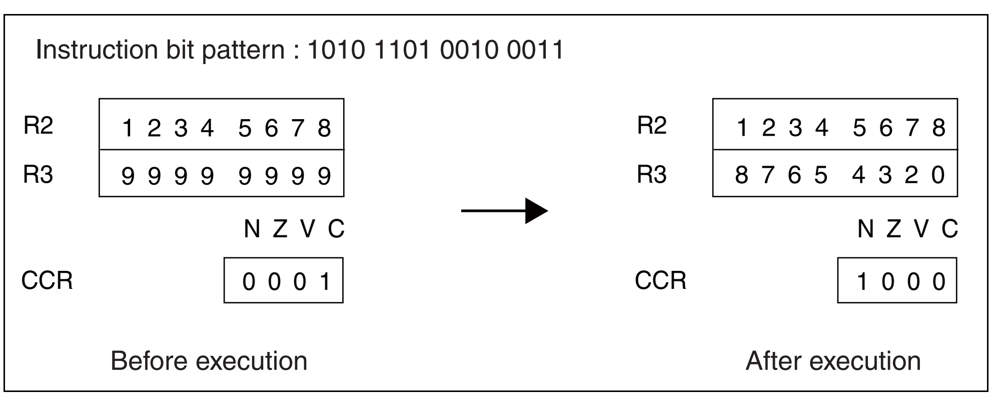
Execution cycles: 1 cycle

Instruction format:

MSB ... LSB

1	0	1	0	1	1	0	1	Rj	Ri

Example: SUBC R2, R3



7.10 SUBN (Subtract Word Data in Source Register from Destination Register)

Subtracts the word data in "Rj" from the word data in "Ri", stores results to "Ri" without changing the flag settings.

■ SUBN (Subtract Word Data in Source Register from Destination Register)

Assembler format: SUBN Rj, Ri

Operation: Ri – Rj → Ri

Flag change:

N	Z	V	C
–	–	–	–

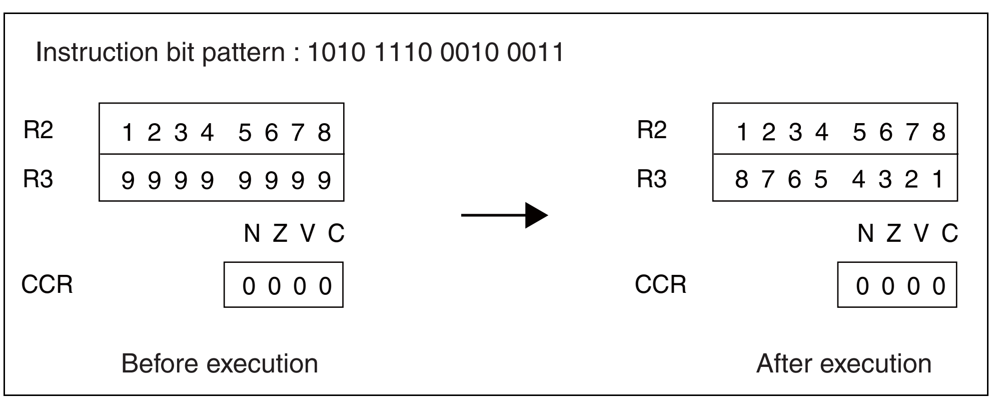
N, Z, V, and C: Unchanged

Execution cycles: 1 cycle

Instruction format:

MSB									LSB
1	0	1	0	1	1	1	0	Rj	Ri

Example: SUBN R2, R3



7.11 CMP (Compare Word Data in Source Register and Destination Register)

Subtracts the word data in "Rj" from the word data in "Ri", places results in the condition code register (CCR).

■ CMP (Compare Word Data in Source Register and Destination Register)

Assembler format: CMP Rj, Ri

Operation: Ri – Rj

Flag change:

N	Z	V	C
C	C	C	C

N : Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z : Set when the operation result is "0", cleared otherwise.

V : Set when an overflow has occurred as a result of the operation, cleared otherwise.

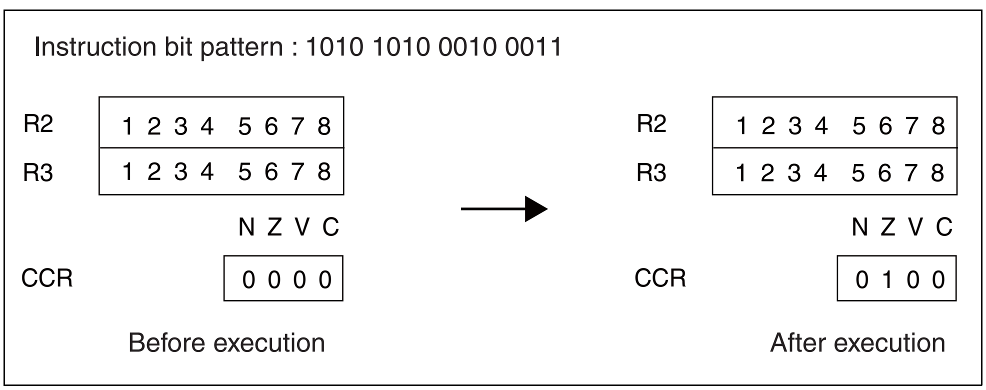
C : Set when a borrow has occurred as a result of the operation, cleared otherwise.

Execution cycles: 1 cycle

Instruction format:

MSB									LSB
1	0	1	0	1	0	1	0	Rj	Ri

Example: CMP R2, R3



7.12 CMP (Compare Immediate Data of Source Register and Destination Register)

Subtracts the result of the higher 28 bits of 4-bit immediate data with zero extension from the word data in "Ri", places results in the condition code register (CCR).

■ CMP (Compare Immediate Data of Source Register and Destination Register)

Assembler format: CMP #i4, Ri

Operation: Ri – extu(i4)

Flag change:

N	Z	V	C
C	C	C	C

N : Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z : Set when the operation result is "0", cleared otherwise.

V : Set when an overflow has occurred as a result of the operation, cleared otherwise.

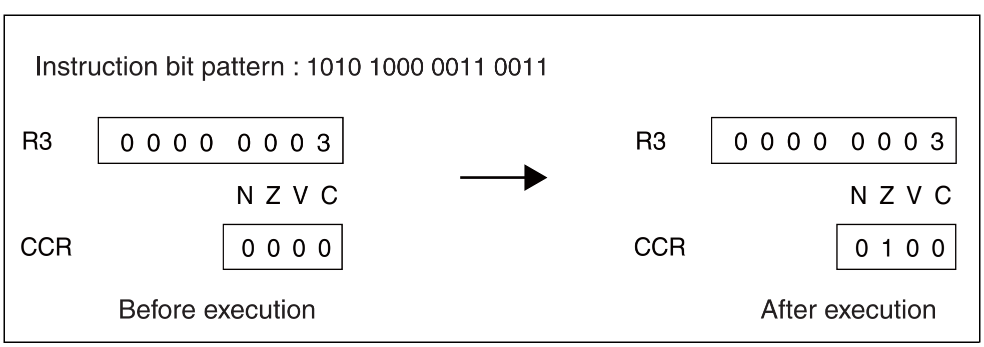
C : Set when a carry has occurred as a result of the operation, cleared otherwise.

Execution cycles: 1 cycle

Instruction format:

MSB									LSB
1	0	1	0	1	0	0	0	i4	Ri

Example: CMP #3, R3



7.13 CMP2 (Compare Immediate Data and Destination Register)

Subtracts the result of the higher 28 bits of 4-bit immediate(from -16 to -1) data with minus extension from the word data in "Ri", places results in the condition code register (CCR).

■ CMP2 (Compare Immediate Data and Destination Register)

Assembler format: CMP2 #i4, Ri

Operation: Ri – extn(i4)

Flag change:

N	Z	V	C
C	C	C	C

N : Set when the MSB of the operation result is "1",cleared when the MSB is "0".

Z : Set when the operation result is "0", cleared otherwise.

V : Set when an overflow has occurred as a result of the operation, cleared otherwise.

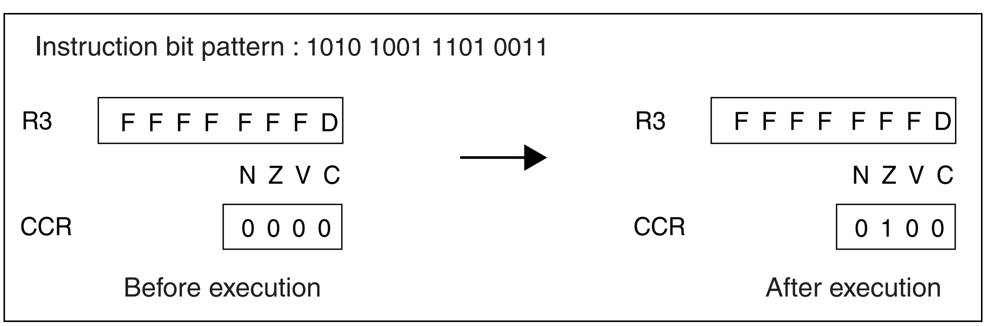
C : Set when a carry has occurred as a result of the operation, cleared otherwise.

Execution cycles: 1 cycle

Instruction format:

MSB									LSB
1	0	1	0	1	0	0	1	i4	Ri

Example: CMP2 #–3, R3



7.14 AND (And Word Data of Source Register to Destination Register)

Takes the logical AND of the word data in "Rj" and the word data in "Ri", stores the results to "Ri".

■ AND (And Word Data of Source Register to Destination Register)

Assembler format: AND Rj, Ri

Operation: Ri and Rj → Ri

Flag change:

N	Z	V	C
C	C	–	–

N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is "0", cleared otherwise.

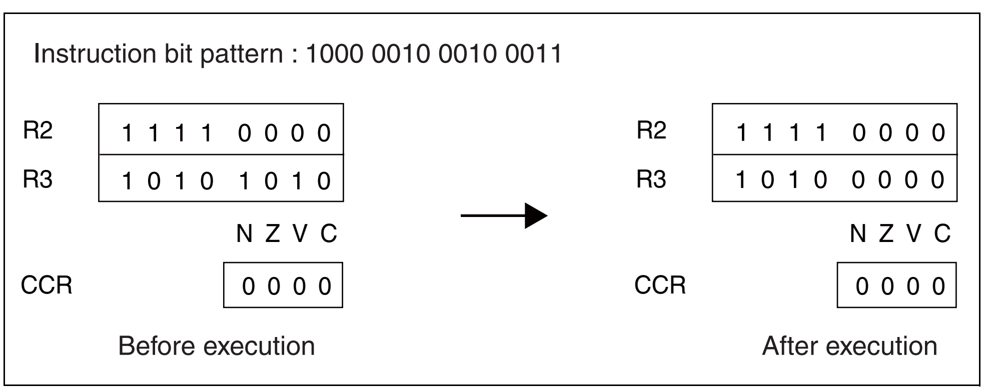
V and C: Unchanged

Execution cycles: 1 cycle

Instruction format:

MSB									LSB
1	0	0	0	0	0	1	0	Rj	Ri

Example: AND R2, R3



7.15 AND (And Word Data of Source Register to Data in Memory)

Takes the logical AND of the word data at memory address "Ri" and the word data in "Rj", stores the results to the memory address corresponding to "Ri".
The CPU will not accept hold requests between the memory read operation and the memory write operation of this request.

■ AND (And Word Data of Source Register to Data in Memory)

Assembler format: AND Rj, @Ri

Operation: (Ri) and Rj → (Ri)

Flag change:

N	Z	V	C
C	C	–	–

N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is "0", cleared otherwise.

V and C: Unchanged

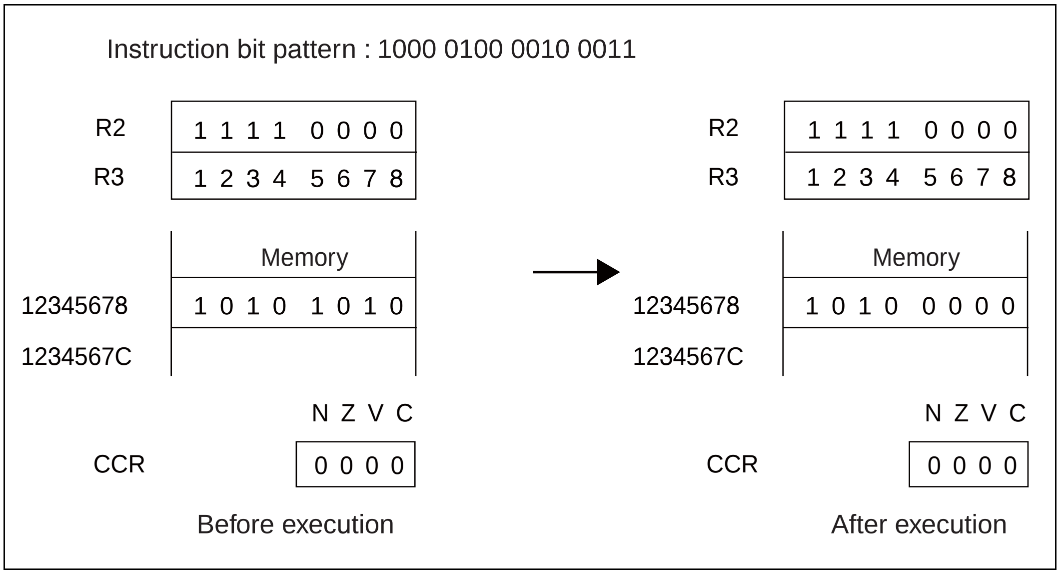
Execution cycles: 1 + 2a cycles

Instruction format:

MSB LSB

1	0	0	0	0	1	0	0	Rj	Ri

Example: AND R2, @R3



7.16 ANDH (And Half-word Data of Source Register to Data in Memory)

Takes the logical AND of the half-word data at memory address "Ri" and the half-word data in "Rj", stores the results to the memory address corresponding to "Ri".
The CPU will not accept hold requests between the memory read operation and the memory write operation of this request.

■ ANDH (And Half-word Data of Source Register to Data in Memory)

Assembler format: ANDH Rj, @Ri

Operation: (Ri) and Rj → (Ri)

Flag change:

N	Z	V	C
C	C	–	–

N: Set when the MSB (bit 15) of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is "0", cleared otherwise.

V and C: Unchanged

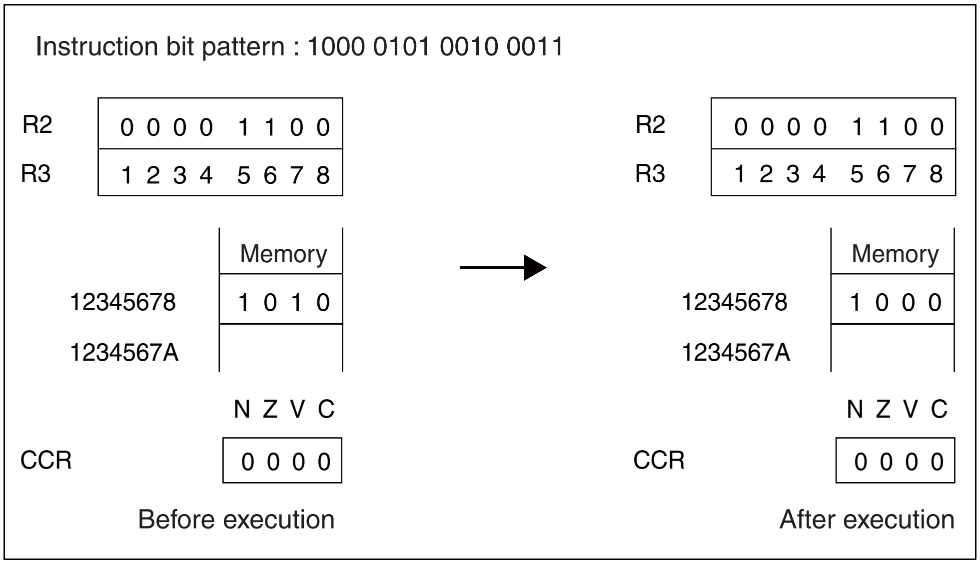
Execution cycles: 1 + 2a cycles

Instruction format:

MSB ... LSB

1	0	0	0	0	1	0	1	Rj	Ri

Example: ANDH R2, @R3



7.17 ANDB (And Byte Data of Source Register to Data in Memory)

Takes the logical AND of the byte data at memory address "Ri" and the byte data in "Rj", stores the results to the memory address corresponding to "Ri".
The CPU will not accept hold requests between the memory read operation and the memory write operation of this request.

■ ANDB (And Byte Data of Source Register to Data in Memory)

Assembler format: ANDB Rj, @Ri

Operation: (Ri) and Rj → (Ri)

Flag change:

N	Z	V	C
C	C	–	–

N: Set when the MSB (bit 7) of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is "0", cleared otherwise.

V and C: Unchanged

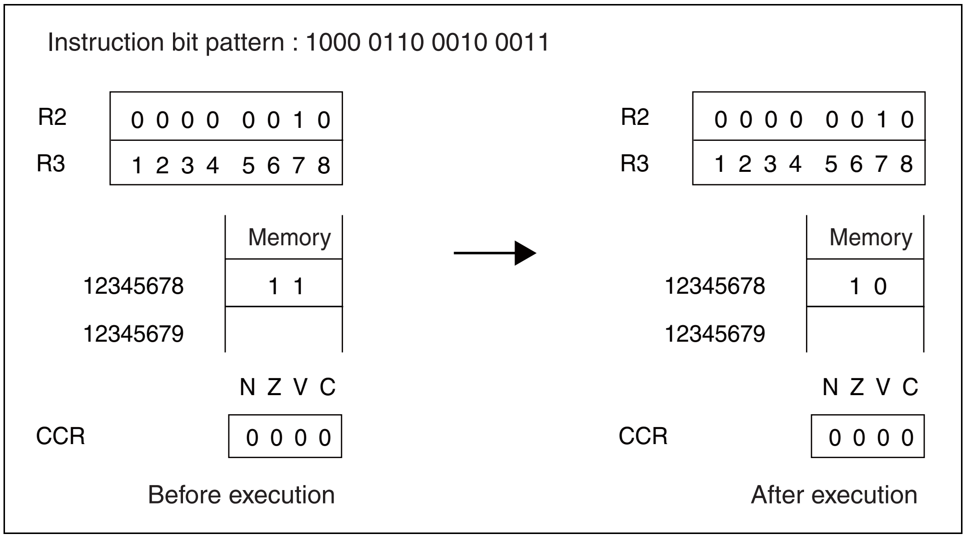
Execution cycles: 1 + 2a cycles

Instruction format:

MSB ... LSB

1	0	0	0	0	1	1	0	Rj	Ri

Example: ANDB R2, @R3



7.18 OR (Or Word Data of Source Register to Destination Register)

Takes the logical OR of the word data in "Ri" and the word data in "Rj", stores the results to "Ri".

■ OR (Or Word Data of Source Register to Destination Register)

Assembler format: OR Rj, Ri

Operation: Ri or Rj → Ri

Flag change:

N	Z	V	C
C	C	–	–

N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is "0", cleared otherwise.

V and C: Unchanged

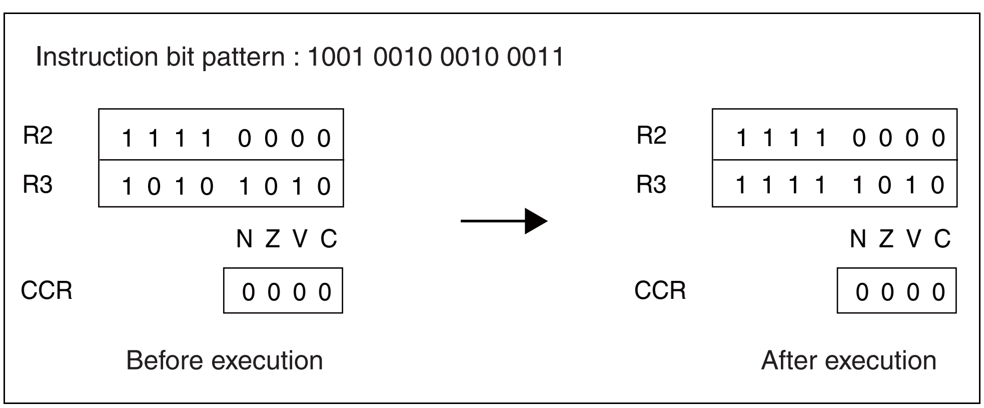
Execution cycles: 1 cycle

Instruction format:

MSB … LSB

1	0	0	1	0	0	1	0	Rj	Ri

Example: OR R2, R3



7.19 OR (Or Word Data of Source Register to Data in Memory)

Takes the logical OR of the word data at memory address "Ri" and the word data in "Rj", stores the results to the memory address corresponding to "Ri".
The CPU will not accept hold requests between the memory read operation and the memory write operation of this request.

■ OR (Or Word Data of Source Register to Data in Memory)

Assembler format: OR Rj, @Ri

Operation: (Ri) or Rj → (Ri)

Flag change:

N	Z	V	C
C	C	–	–

N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is "0", cleared otherwise.

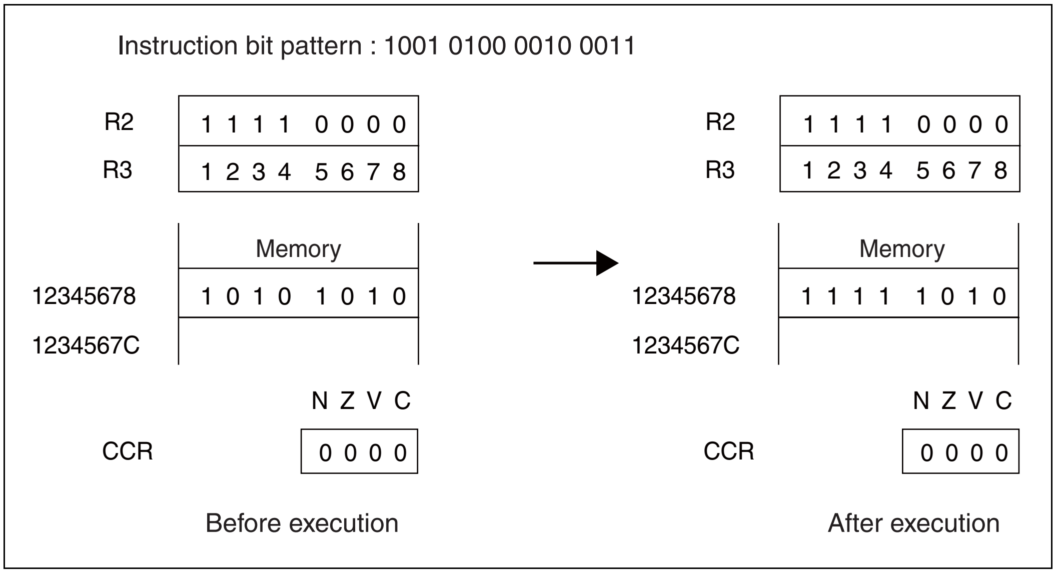
V and C: Unchanged

Execution cycles: 1 + 2a cycles

Instruction format:

MSB									LSB
1	0	0	1	0	1	0	0	Rj	Ri

Example: OR R2, @R3



7.20 ORH (Or Half-word Data of Source Register to Data in Memory)

Takes the logical OR of the half-word data at memory address "Ri" and the half-word data in "Rj", stores the results to the memory address corresponding to "Ri".
The CPU will not accept hold requests between the memory read operation and the memory write operation of this request.

■ ORH (Or Half-word Data of Source Register to Data in Memory)

Assembler format: ORH Rj, @Ri

Operation: (Ri) or Rj → (Ri)

Flag change:

N	Z	V	C
C	C	–	–

N: Set when the MSB (bit 15) of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is "0", cleared otherwise.

V and C: Unchanged

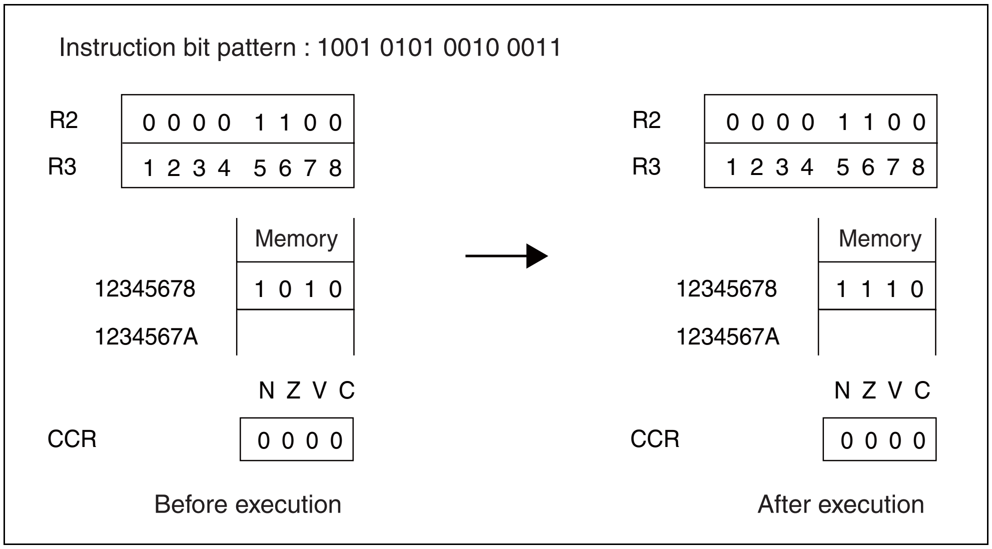
Execution cycles: 1 + 2a cycles

Instruction format:

MSB → LSB

1	0	0	1	0	1	0	1	Rj	Ri

Example: ORH R2, @R3



7.21 ORB (Or Byte Data of Source Register to Data in Memory)

Takes the logical OR of the byte data at memory address "Ri" and the byte data in "Rj", stores the results to the memory address corresponding to "Ri".
The CPU will not accept hold requests between the memory read operation and the memory write operation of this request.

■ ORB (Or Byte Data of Source Register to Data in Memory)

Assembler format: ORB Rj, @Ri

Operation: (Ri) or Rj → (Ri)

Flag change:

N	Z	V	C
C	C	–	–

N: Set when the MSB (bit 7) of the operation result is "1", cleared when the MSB is "0".
Z: Set when the operation result is "0", cleared otherwise.
V and C: Unchanged

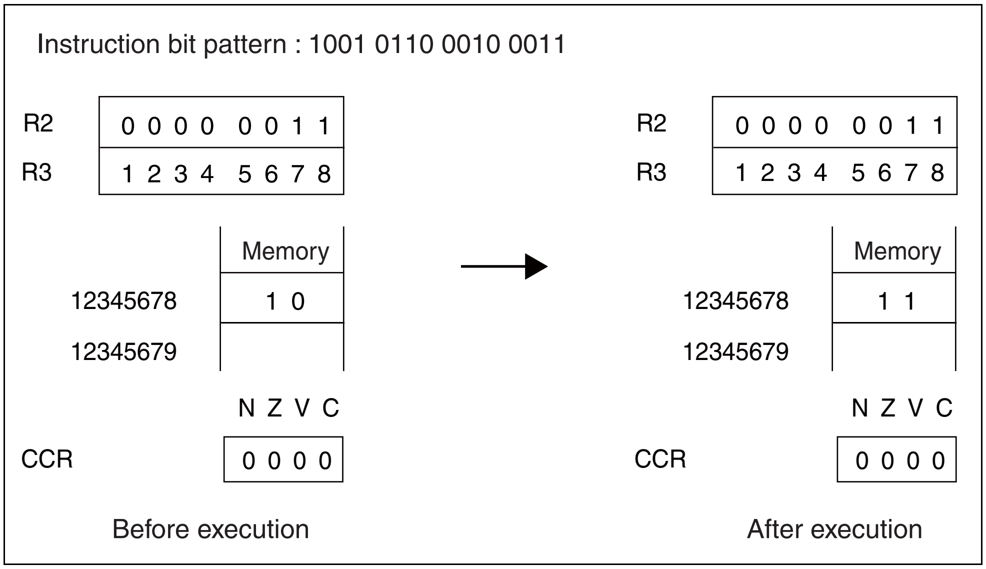
Execution cycles: 1 + 2a cycles

Instruction format:

MSB ... LSB

1	0	0	1	0	1	1	0	Rj	Ri

Example: ORB R2, @R3



7.22 EOR (Exclusive Or Word Data of Source Register to Destination Register)

Takes the logical exclusive OR of the word data in "Ri" and the word data in "Rj", stores the results to "Ri".

■ EOR (Exclusive Or Word Data of Source Register to Destination Register)

Assembler format: EOR Rj, Ri

Operation: Ri eor Rj → (Ri)

Flag change:

N	Z	V	C
C	C	–	–

N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is "0", cleared otherwise.

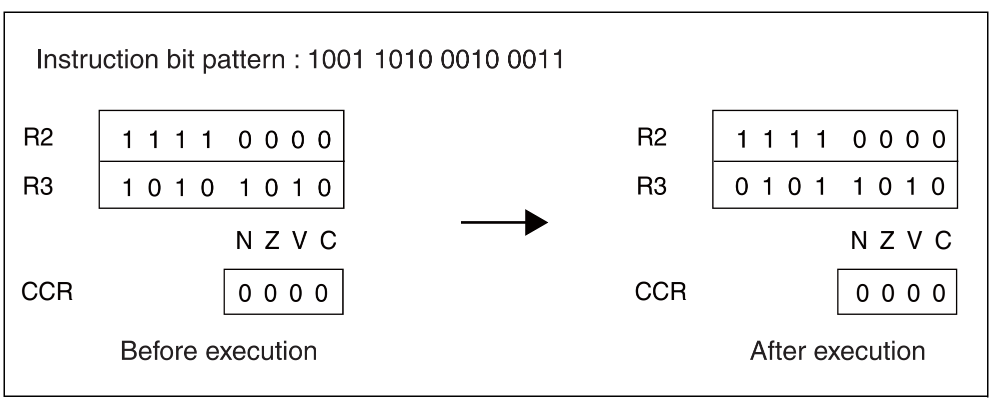
V and C: Unchanged

Execution cycles: 1 cycle

Instruction format:

MSB									LSB
1	0	0	1	1	0	1	0	Rj	Ri

Example: EOR R2, R3



7.23 EOR (Exclusive Or Word Data of Source Register to Data in Memory)

Takes the logical exclusive OR of the word data at memory address "Ri" and the word data in "Rj", stores the results to the memory address corresponding to "Ri".
The CPU will not accept hold requests between the memory read operation and the memory write operation of this request.

■ EOR (Exclusive Or Word Data of Source Register to Data in Memory)

Assembler format: EOR Rj, @Ri

Operation: (Ri) eor Rj → (Ri)

Flag change:

N	Z	V	C
C	C	–	–

N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is "0", cleared otherwise.

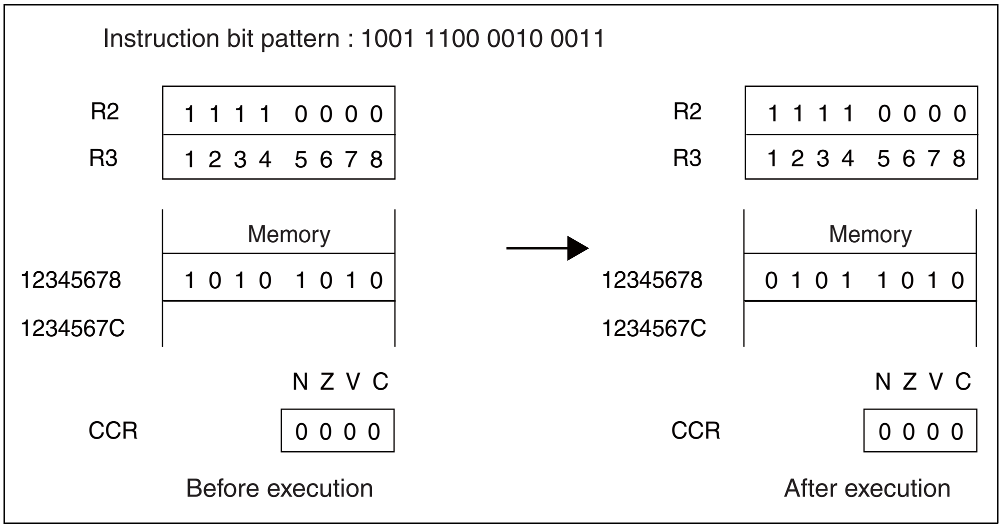
V and C: Unchanged

Execution cycles: 1 + 2a cycles

Instruction format:

MSB									LSB
1	0	0	1	1	1	0	0	Rj	Ri

Example: EOR R2, @R3



7.24 EORH (Exclusive Or Half-word Data of Source Register to Data in Memory)

Takes the logical exclusive OR of the half-word data at memory address "Ri" and the half-word data in "Rj", stores the results to the memory address corresponding to "Ri". The CPU will not accept hold requests between the memory read operation and the memory write operation of this request.

■ EORH (Exclusive Or Half-word Data of Source Register to Data in Memory)

Assembler format: EORH Rj, @Ri

Operation: (Ri) eor Rj → (Ri)

Flag change:

N	Z	V	C
C	C	–	–

N: Set when the MSB (bit 15) of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is "0", cleared otherwise.

V and C: Unchanged

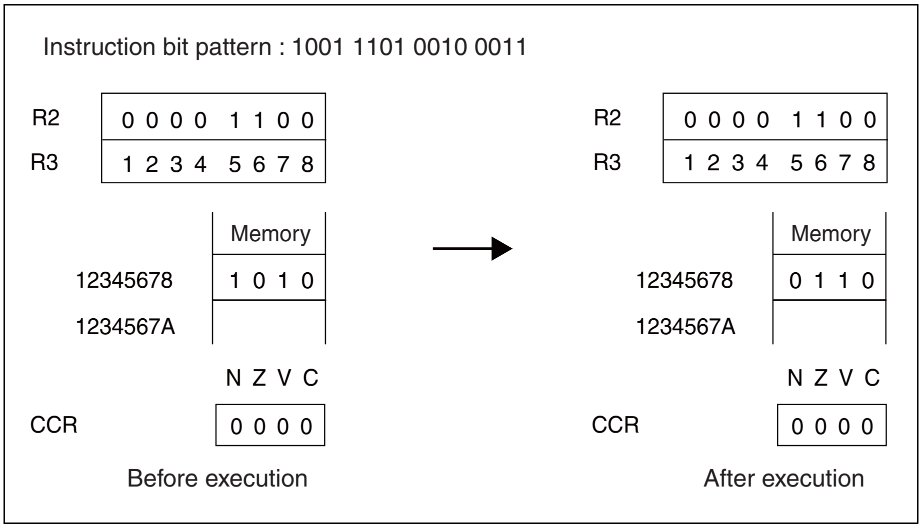
Execution cycles: 1 + 2a cycles

Instruction format:

MSB LSB

1	0	0	1	1	1	0	1	Rj	Ri

Example: EORH R2, @R3



7.25 EORB (Exclusive Or Byte Data of Source Register to Data in Memory)

Takes the logical exclusive OR of the byte data at memory address "Ri" and the byte data in "Rj", stores the results to the memory address corresponding to "Ri".
The CPU will not accept hold requests between the memory read operation and the memory write operation of this request.

■ EORB (Exclusive Or Byte Data of Source Register to Data in Memory)

Assembler format: EORB Rj, @Ri

Operation: (Ri) eor Rj → (Ri)

Flag change:

N	Z	V	C
C	C	–	–

N: Set when the MSB (bit 7) of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is "0", cleared otherwise.

V and C: Unchanged

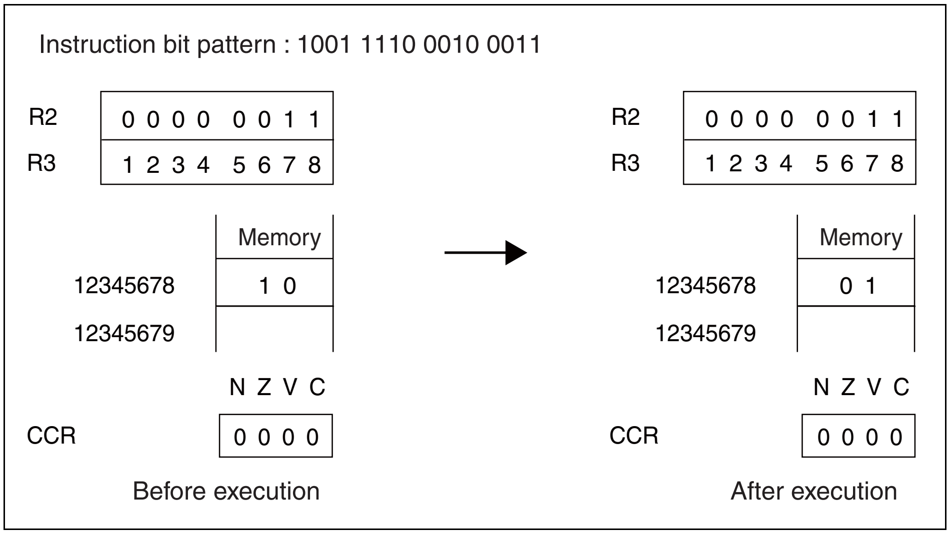
Execution cycles: 1 + 2a cycles

Instruction format:

MSB … LSB

1	0	0	1	1	1	1	0	Rj	Ri

Example: EORB R2, @R3



7.26 BANDL (And 4-bit Immediate Data to Lower 4 Bits of Byte Data in Memory)

Takes the logical AND of the 4-bit immediate data and the lower 4 bits of byte data at memory "Ri", stores the results to the memory address corresponding to "Ri".
The CPU will not accept hold requests between the memory read operation and the memory write operation of this request.

■ BANDL (And 4-bit Immediate Data to Lower 4 Bits of Byte Data in Memory)

Assembler format: BANDL #u4, @Ri

Operation: {F0$_H$ + u4} and (Ri) → (Ri) [Operation uses lower 4 bits only]

Flag change:

N	Z	V	C
–	–	–	–

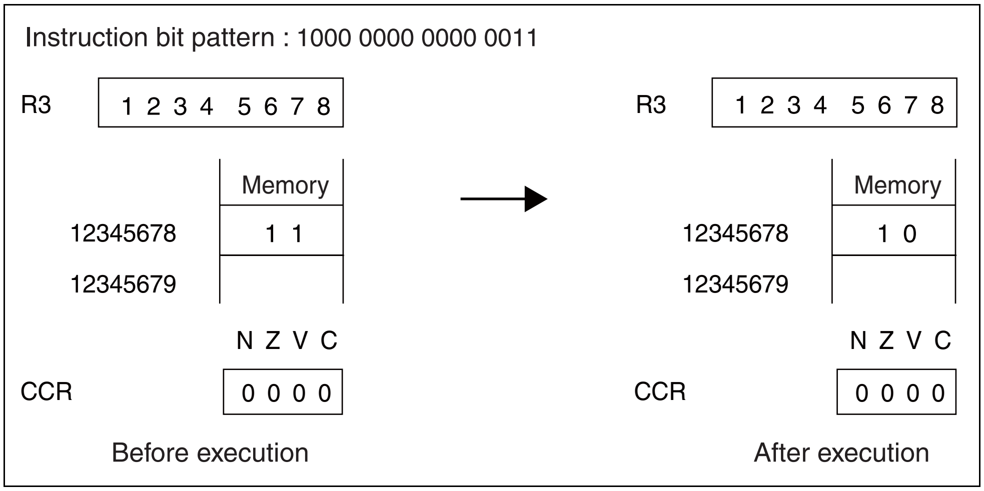
N, Z, V, and C: Unchanged

Execution cycles: 1 + 2a cycles

Instruction format:

MSB									LSB
1	0	0	0	0	0	0	0	u4	Ri

Example: BANDL #0, @R3



7.27 BANDH (And 4-bit Immediate Data to Higher 4 Bits of Byte Data in Memory)

Takes the logical AND of the 4-bit immediate data and the higher 4 bits of byte data at memory "Ri", stores the results to the memory address corresponding to "Ri".
The CPU will not accept hold requests between the memory read operation and the memory write operation of this request.

■ BANDH (And 4-bit Immediate Data to Higher 4 Bits of Byte Data in Memory)

Assembler format: BANDH #u4, @Ri

Operation: {u4 < < 4 + F_H} and (Ri) → (Ri) [Operation uses higher 4 bits only]

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

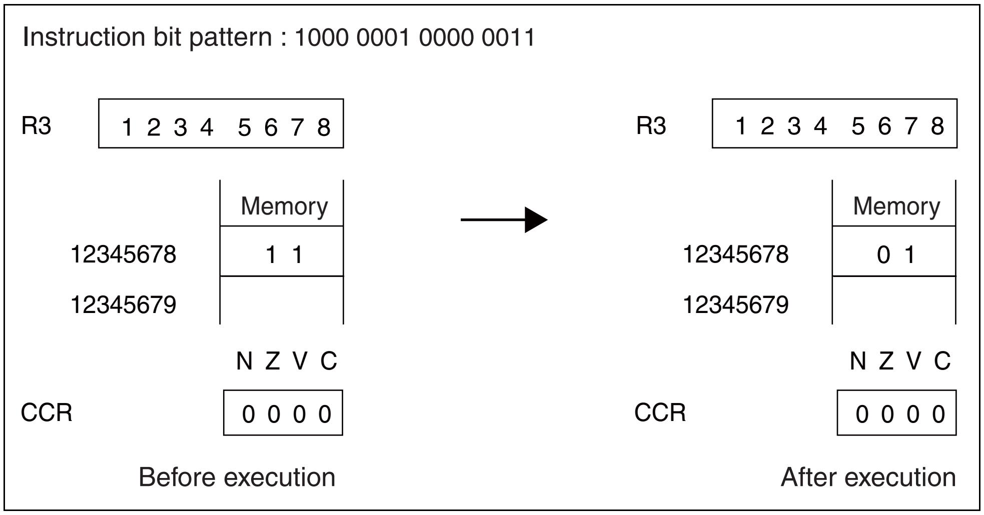
Execution cycles: 1 + 2a cycles

Instruction format:

MSB ... LSB

1	0	0	0	0	0	0	1	u4	Ri

Example: BANDH #0, @R3



7.28 BORL (Or 4-bit Immediate Data to Lower 4 Bits of Byte Data in Memory)

Takes the logical OR of the 4-bit immediate data and the lower 4 bits of byte data at memory address "Ri", stores the results to the memory address corresponding to "Ri". The CPU will not accept hold requests between the memory read operation and the memory write operation of this request.

■ BORL (Or 4-bit Immediate Data to Lower 4 Bits of Byte Data in Memory)

Assembler format: BORL #u4, @Ri

Operation: u4 or (Ri) → (Ri) [Operation uses lower 4 bits only]

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

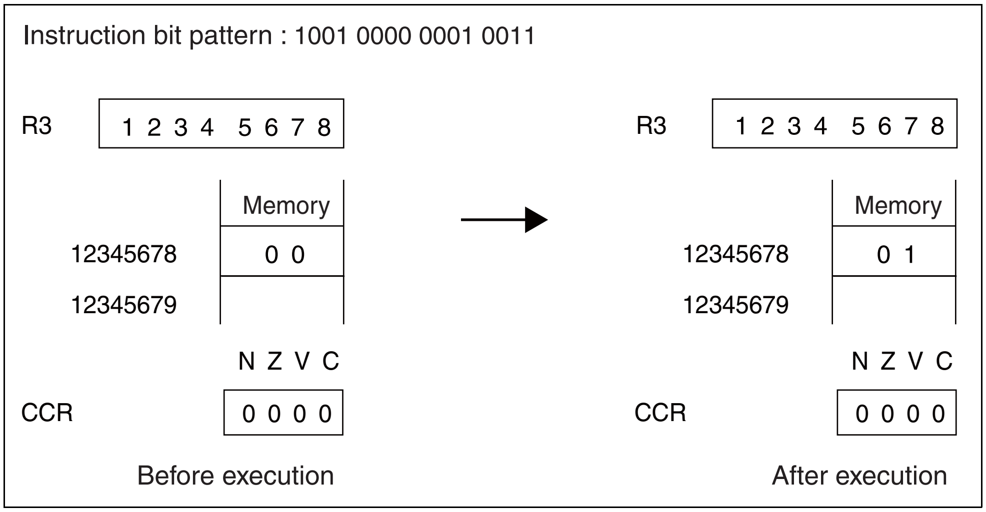
Execution cycles: 1 + 2a cycles

Instruction format:

MSB ... LSB

1	0	0	1	0	0	0	0	u4	Ri

Example: BORL #1, @R3



7.29 BORH (Or 4-bit Immediate Data to Higher 4 Bits of Byte Data in Memory)

Takes the logical OR of the 4-bit immediate data and the higher 4 bits of byte data at memory address "Ri", stores the results to the memory address corresponding to "Ri". The CPU will not accept hold requests between the memory read operation and the memory write operation of this request.

■ BORH (Or 4-bit Immediate Data to Higher 4 Bits of Byte Data in Memory)

Assembler format: BORH #u4, @Ri

Operation: {u4 < < 4} or (Ri) → (Ri) [Operation uses higher 4 bits only]

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

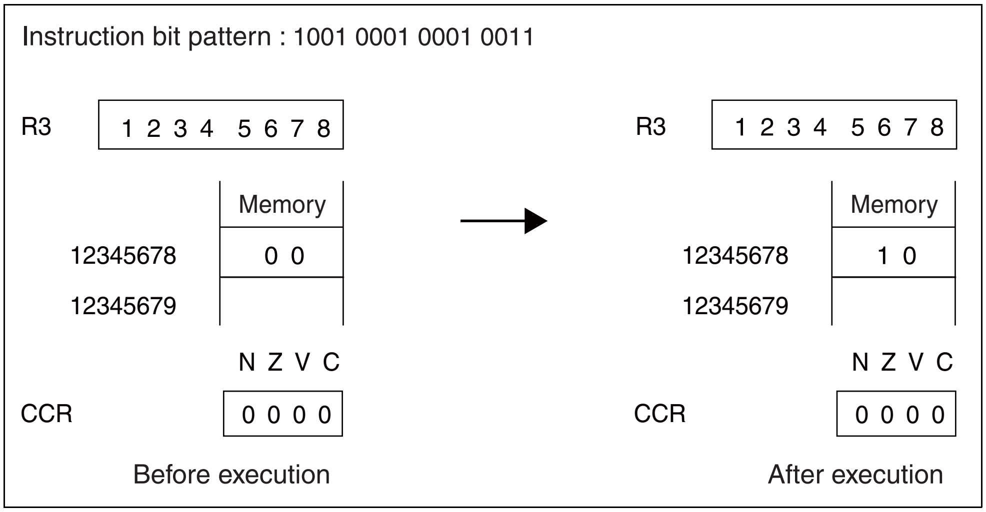
Execution cycles: 1 + 2a cycles

Instruction format:

MSB (left) to LSB (right):

1	0	0	1	0	0	0	1	u4	Ri

Example: BORH #1, @R3



7.30 BEORL (Eor 4-bit Immediate Data to Lower 4 Bits of Byte Data in Memory)

Takes the logical exclusive OR of the 4-bit immediate data and the lower 4 bits of byte data at memory address "Ri", stores the results to the memory address corresponding to "Ri".
The CPU will not accept hold requests between the memory read operation and the memory write operation of this request.

■ BEORL (Eor 4-bit Immediate Data to Lower 4 Bits of Byte Data in Memory)

Assembler format: BEORL #u4, @Ri

Operation: u4 eor (Ri) → (Ri) [Operation uses lower 4 bits only]

Flag change:

N	Z	V	C
–	–	–	–

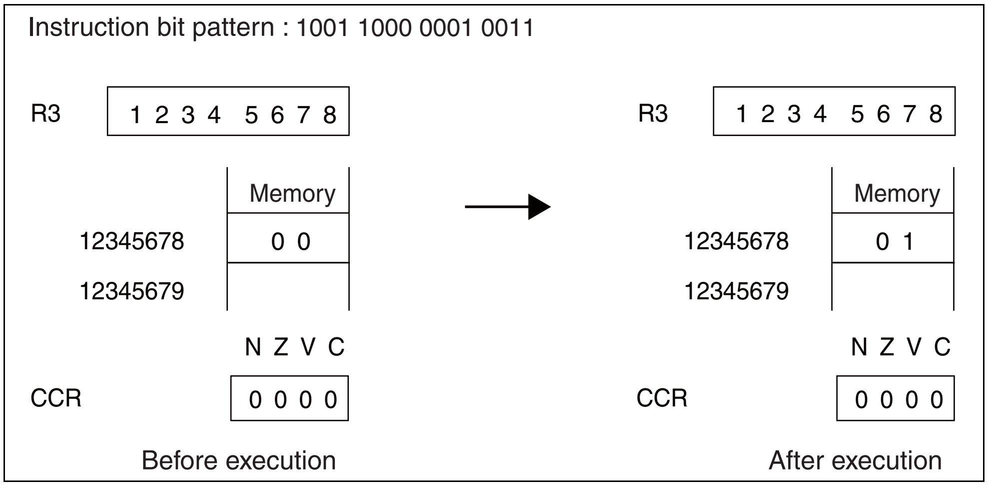
N, Z, V, and C: Unchanged

Execution cycles: 1 + 2a cycles

Instruction format:

MSB									LSB
1	0	0	1	1	0	0	0	u4	Ri

Example: BEORL #1, @R3



7.31 BEORH (Eor 4-bit Immediate Data to Higher 4 Bits of Byte Data in Memory)

Takes the logical exclusive OR of the 4-bit immediate data and the higher 4 bits of byte data at memory address "Ri", stores the results to the memory address corresponding to "Ri".
The CPU will not accept hold requests between the memory read operation and the memory write operation of this request.

■ BEORH (Eor 4-bit Immediate Data to Higher 4 Bits of Byte Data in Memory)

Assembler format: BEORH #u4, @Ri

Operation: {u4 < < 4} eor (Ri) → (Ri) [Operation uses higher 4 bits only]

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

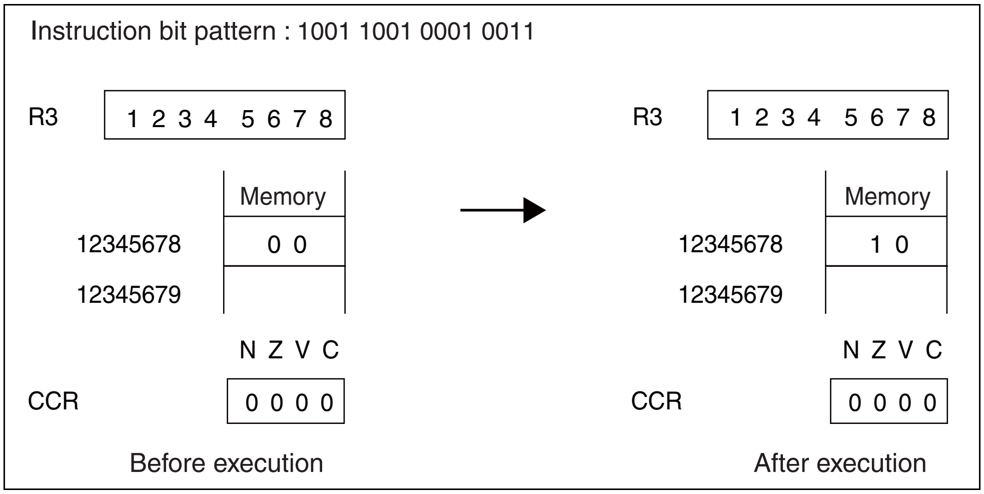
Execution cycles: 1 + 2a cycles

Instruction format:

MSB ... LSB

1	0	0	1	1	0	0	1	u4	Ri

Example: BEORH #1, @R3



7.32 BTSTL (Test Lower 4 Bits of Byte Data in Memory)

Takes the logical AND of the 4-bit immediate data and the lower 4 bits of byte data at memory address "Ri", places the results in the condition code register (CCR).

■ BTSTL (Test Lower 4 Bits of Byte Data in Memory)

Assembler format: BTSTL #u4, @Ri

Operation: u4 and (Ri) [Test uses lower 4 bits only]

Flag change:

N	Z	V	C
0	C	–	–

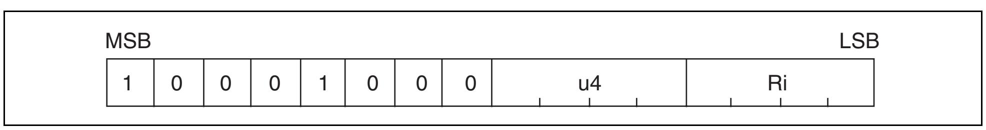
N: Cleared

Z: Set when the operation result is "0", cleared otherwise.

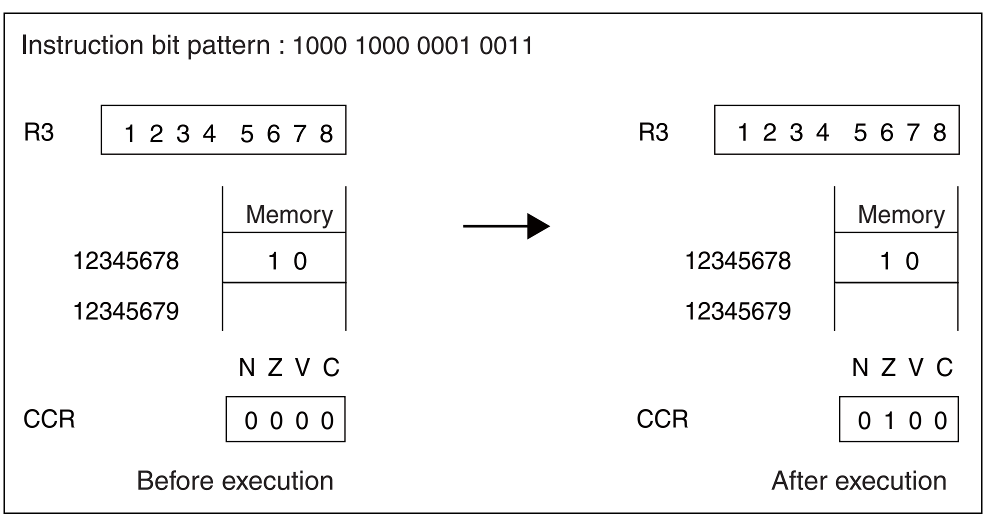
V and C: Unchanged

Execution cycles: 2+a cycles

Instruction format:



Example: BTSTL #1, @R3



7.33 BTSTH (Test Higher 4 Bits of Byte Data in Memory)

Takes the logical AND of the 4-bit immediate data and the higher 4 bits of byte data at memory address "Ri", places the results in the condition code register (CCR).

■ BTSTH (Test Higher 4 Bits of Byte Data in Memory)

Assembler format: BTSTH #u4, @Ri

Operation: {u4 < < 4} and (Ri)   [Test uses higher 4 bits only]

Flag change:

N	Z	V	C
C	C	–	–

N: Set when the MSB (bit 7) of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is "0", cleared otherwise.

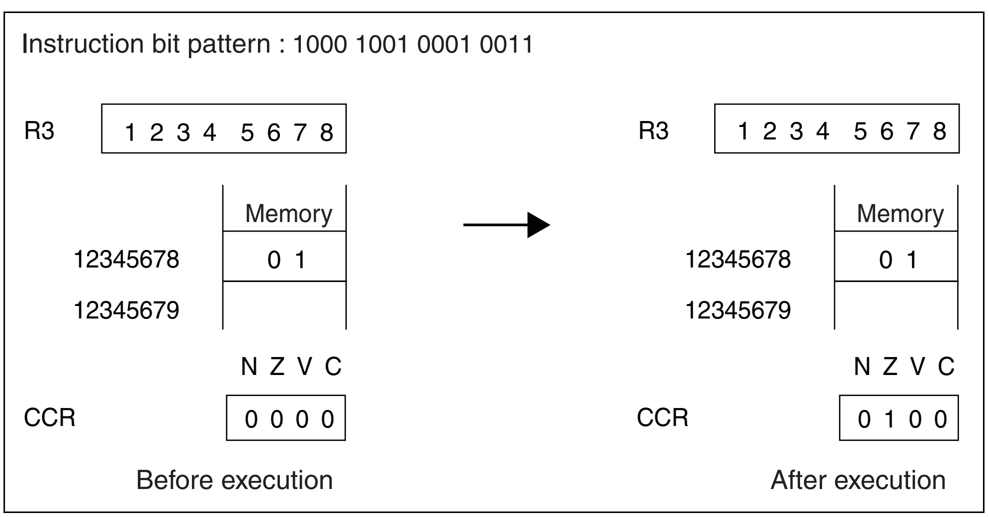
V and C: Unchanged

Execution cycles: 2 + a cycles

Instruction format:

MSB									LSB
1	0	0	0	1	0	0	1	u4	Ri

Example: BTSTH #1, @R3



7.34 MUL (Multiply Word Data)

Multiplies the word data in "Rj" by the word data in "Ri" as signed numbers, and stores the resulting signed 64-bit data with the high word in the multiplication/division register (MDH), and the low word in the multiplication/division register (MDL).

■ MUL (Multiply Word Data)

Assembler format: MUL Rj, Ri

Operation: Rj × Ri → MDH, MDL

Flag change:

N	Z	V	C
C	C	C	–

N: Set when the MSB of the "MDL" of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is "0", cleared otherwise.

V: Cleared when the operation result is in the range -2147483648 to 2147483647, set otherwise.

C: Unchanged

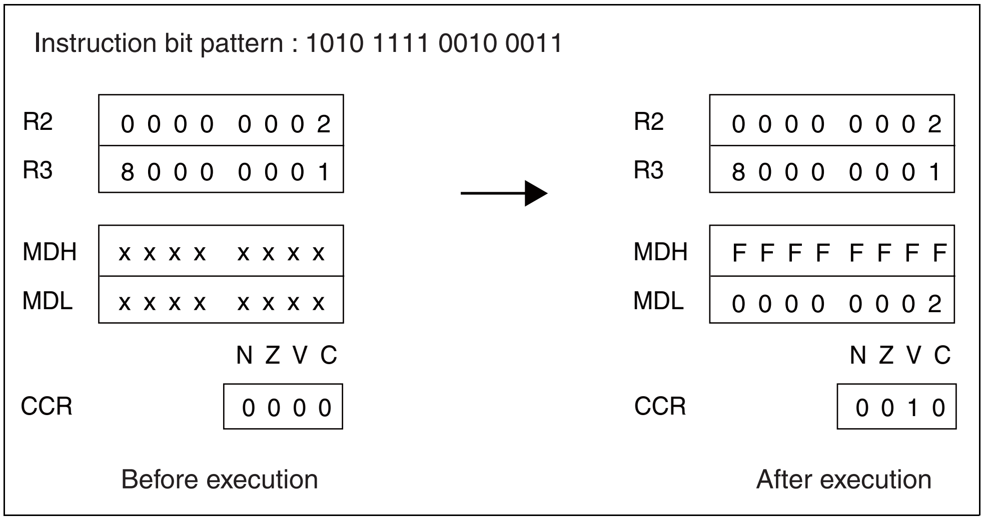
Execution cycles: 5 cycles

Instruction format:

MSB ... LSB

1	0	1	0	1	1	1	1	Rj	Ri

Example: MUL R2, R3



7.35 MULU (Multiply Unsigned Word Data)

Multiplies the word data in "Rj" by the word data in "Ri" as unsigned numbers, and stores the resulting unsigned 64-bit data with the high word in the multiplication/ division register (MDH), and the low word in the multiplication/division register (MDL).

■ MULU (Multiply Unsigned Word Data)

Assembler format: MULU Rj, Ri

Operation: Rj × Ri → MDH, MDL

Flag change:

N	Z	V	C
C	C	C	–

N: Set when the MSB of the "MDL" of the operation result is "1", cleared when the MSB is "0".

Z: Set when the "MDL" of the operation result is "0", cleared otherwise.

V: Cleared when the operation result is in the range 0 to 4294967295, set otherwise.

C: Unchanged

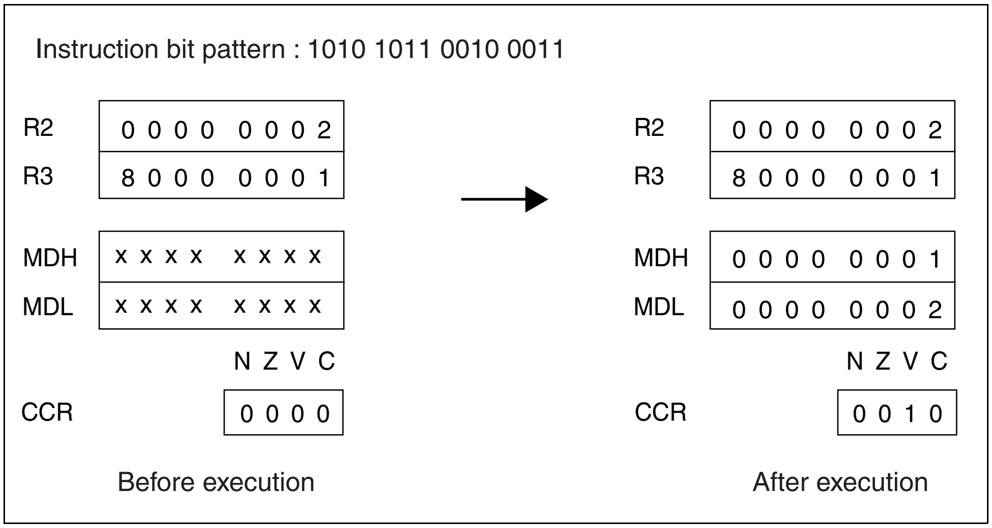
Execution cycles: 5 cycles

Instruction format:

MSB ... LSB

1	0	1	0	1	0	1	1	Rj	Ri

Example: MULU R2, R3



7.36 MULH (Multiply Half-word Data)

Multiplies the half-word data in the lower 16 bits of "Rj" by the half-word data in the lower 16 bits of "Ri" as signed numbers, and stores the resulting signed 32-bit data in the multiplication/division register (MDL).
The multiplication/division register (MDH) is undefined.

■ MULH (Multiply Half-word Data)

Assembler format: MULH Rj, Ri

Operation: Rj × Ri → MDL

Flag change:

N	Z	V	C
C	C	–	–

N: Set when the MSB of the "MDL" of the operation result is "1", cleared when the MSB is "0".

Z: Set when the "MDL" of the operation result is "0", cleared otherwise.

V: Unchanged

C: Unchanged

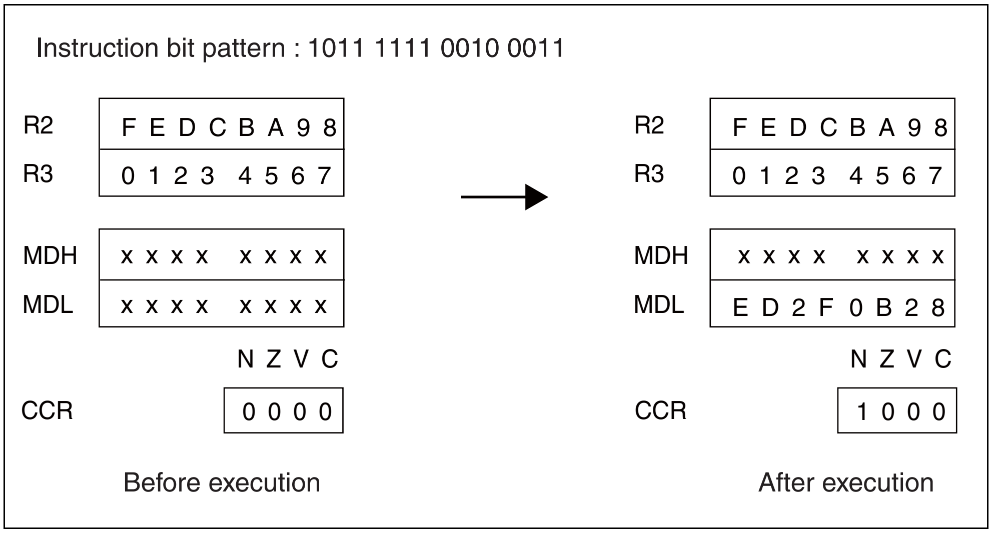
Execution cycles: 3 cycles

Instruction format:

MSB (left) … LSB (right)

1	0	1	1	1	1	1	1	Rj	Ri

Example: MULH R2, R3



7.37 MULUH (Multiply Unsigned Half-word Data)

Multiplies the half-word data in the lower 16 bits of "Rj" by the half-word data in the lower 16 bits of "Ri" as unsigned numbers, and stores the resulting unsigned 32-bit data in the multiplication/division register (MDL).
The multiplication/division register (MDH) is undefined.

■ MULUH (Multiply Unsigned Half-word Data)

Assembler format: MULUH Rj, Ri

Operation: Rj × Ri → MDL

Flag change:

N	Z	V	C
C	C	–	–

N: Set when the MSB of the "MDL" of the operation result is "1", cleared when the MSB is "0".

Z: Set when the "MDL" of the operation result is "0", cleared otherwise.

V: Unchanged

C: Unchanged

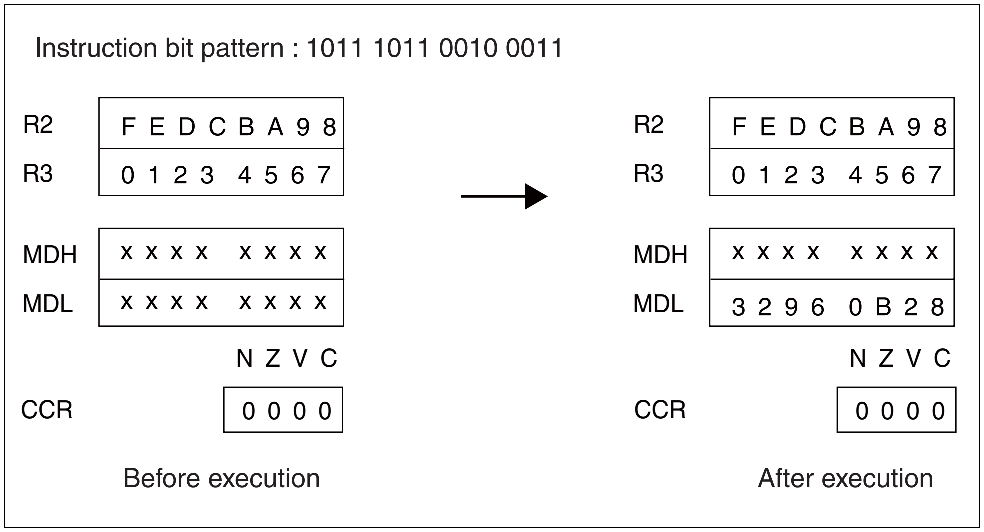
Execution cycles: 3 cycles

Instruction format:

MSB ... LSB

1	0	1	1	1	0	1	1	Rj	Ri

Example: MULUH R2, R3



7.38 DIV0S (Initial Setting Up for Signed Division)

This command is used for signed division in which the multiplication/division register (MDL) contains the dividend and the "Ri" the divisor, with the quotient stored in the "MDL" and the remainder in the multiplication/division register (MDH).
The value of the sign bit in the "MDL" and "Ri" is used to set the "D0" and "D1" flag bits in the system condition code register (SCR).

- **D0: Set when the dividend is negative, cleared when positive.**
- **D1: Set when the divisor and dividend signs are different, cleared when equal.**

The word data in the "MDL" is extended to 64 bits, with the higher word in the "MDH" and the lower word in the "MDL".
To execute signed division, the following instructions are used in combination.
DIV0S, DIV1×32, DIV2, DIV3, DIV4S

■ DIV0S (Initial Setting Up for Signed Division)

Assembler format: DIV0S Ri

Operation:
MDL [31] → D0
MDL [31] eor Ri [31] → D1
exts (MDL) → MDH, MDL

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

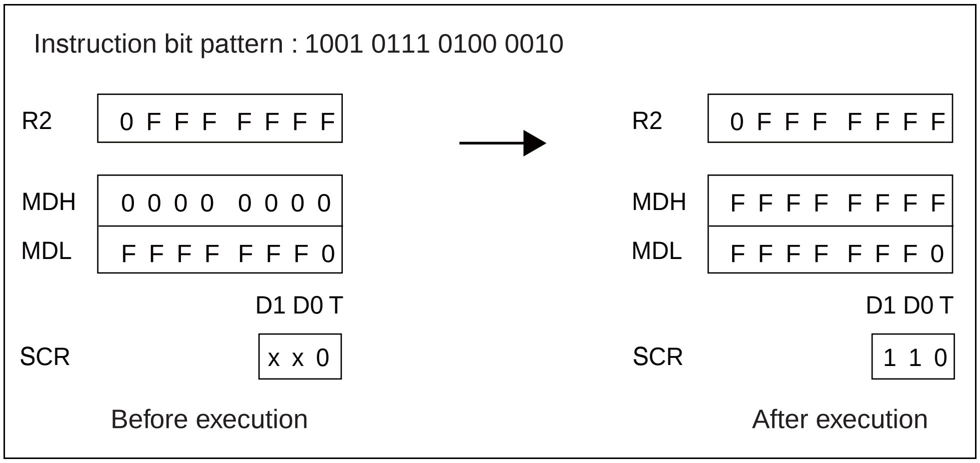
Execution cycles: 1 cycle

Instruction format:

MSB ... LSB

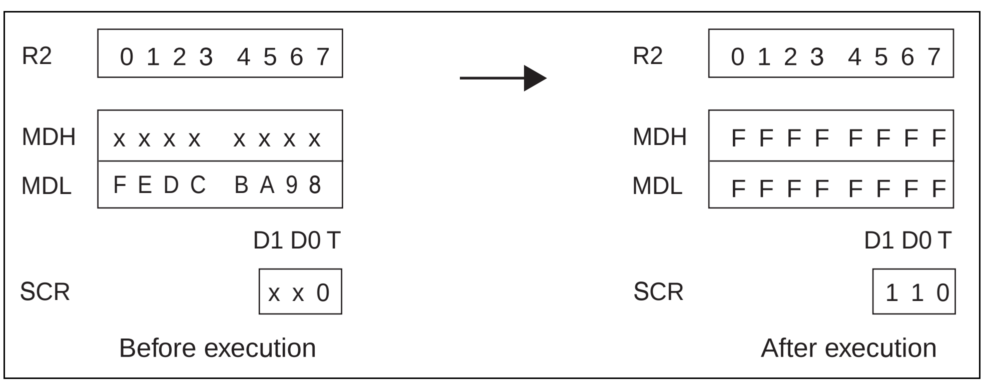
1	0	0	1	0	1	1	1	0	1	0	0	Ri

Example: DIV0S R2



Example: Actual use MDL ÷ R2 = MDL (quotient) ... MDH (remainder), signed calculation

```
DIV0S  R2
DIV1   R2   ┐
DIV1   R2   │
  :     :   ├ 32 DIV1s are arranged
DIV1   R2   ┘
DIV2   R2
DIV3
DIV4S
```



7.39 DIV0U (Initial Setting Up for Unsigned Division)

This command is used for unsigned division in which the multiplication/division register (MDL) contains the dividend and the "Ri" the divisor, with the quotient stored in the "MDL" register and the remainder in the multiplication/division register (MDH).
The "MDH" and bits "D1" and "D0" are cleared to "0".
To execute unsigned division, the instructions are used in combinations such as DIV0U and DIV1 x 32

■ DIV0U (Initial Setting Up for Unsigned Division)

Assembler format: DIV0U Ri

Operation: 0 → D0
0 → D1
0 → MDH

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

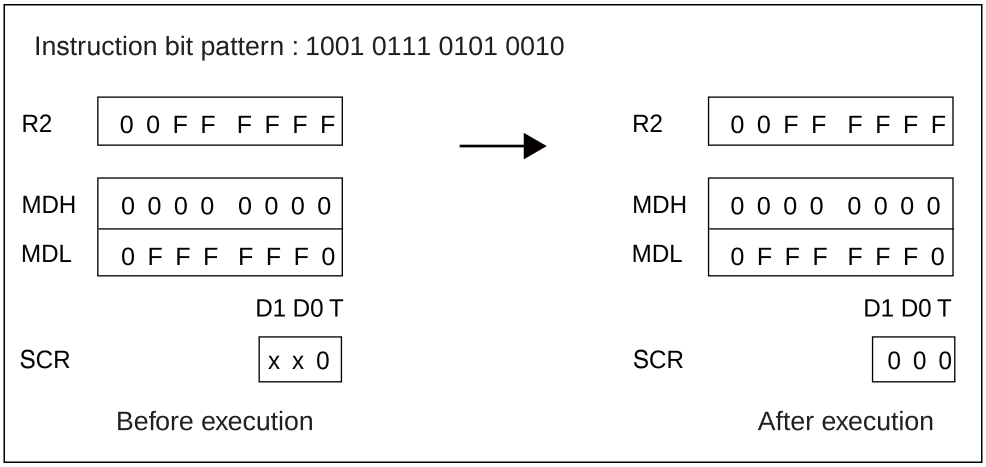
Execution cycles: 1 cycle

Instruction format:

MSB ... LSB

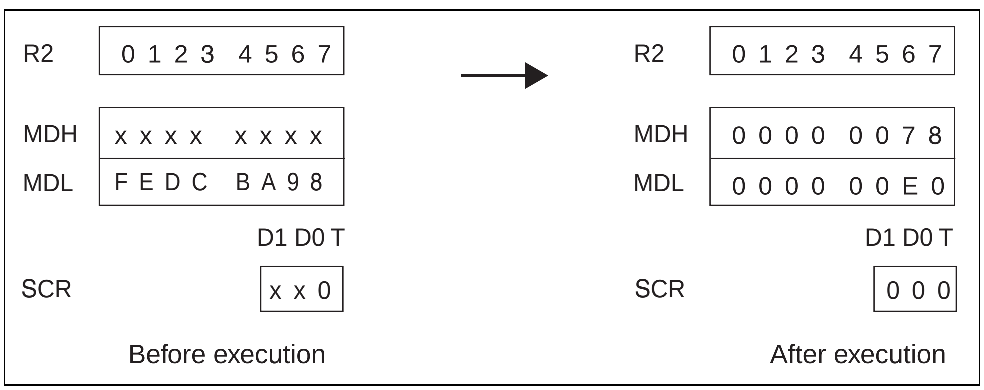
1	0	0	1	0	1	1	1	0	1	0	1	Ri

Example: DIV0U R2



Example: Actual use MDL ÷ R2 = MDL (quotient) ... MDH (remainder), unsigned calculation

```
DIV0U  R2
DIV1   R2  ⎫
DIV1   R2  ⎪
  ⋮     ⋮   ⎬ 32 DIV1s are arranged
DIV1   R2  ⎭
```



7.40 DIV1 (Main Process of Division)

This instruction is used in unsigned division. It should be used in combinations such as DIV0U and DIV1 x 32.

■ DIV1 (Main Process of Division)

Assembler format: DIV1 Ri

Operation:

```
{MDH, MDL} < < = 1
if (D1 = = 1) {
    MDH + Ri → temp
}
else {
    MDH – Ri → temp
}
if ((D0 eor D1 eor C) = = 0) {
    temp → MDH
    1 → MDL [0]
}
```

Flag change:

N	Z	V	C
–	C	–	C

N and V: Unchanged

Z: Set when the result of step division is "0", cleared otherwise. Set according to remainder of division results, not according to quotient.

C: Set when the operation result of step division involves a carry operation, cleared otherwise.

Execution cycles: d cycle(s)

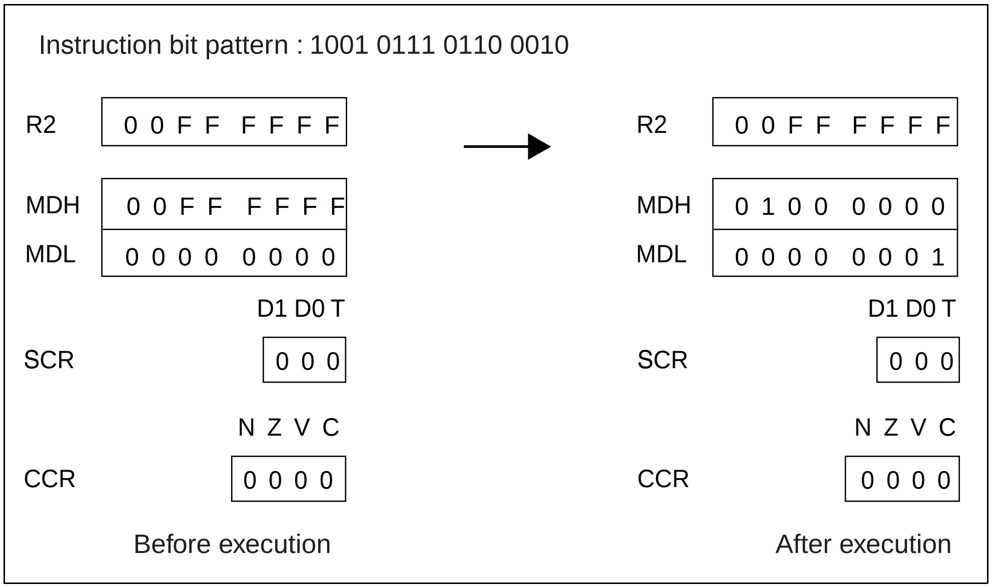
Normally executed within one cycle. However, a 2-cycle interlock is applied if the instruction immediately after is one of the following: MOV MDH, Ri / MOV MDL, Ri / ST Rs, @-R15.

Rs : dedicated register (TBR, RP, USP, SSP, MDH, MDL)

Instruction format:

MSB												LSB
1	0	0	1	0	1	1	1	0	1	1	0	Ri

Example: DIV1 R2



7.41 DIV2 (Correction when Remainder is 0)

This instruction is used in signed division. It should be used in combinations such as DIV0S, DIV1 x 32, DIV2, DIV3 and DIV4S.

■ DIV2 (Correction when Remainder is 0)

Assembler format: DIV2 Ri

Operation:

```
if (D1 = = 1) {
    MDH + Ri → temp
}
else {
    MDH – Ri → temp
}
if (Z == 1) {
    0 → MDH
}
```

Flag change:

N	Z	V	C
–	C	–	C

N and V: Unchanged

Z: Set when the operation result of stepwise division is "0", cleared otherwise. Set according to remainder of division results, not according to quotient.

C: Set when the result of stepwise division involves a carry or borrow operation, cleared otherwise.

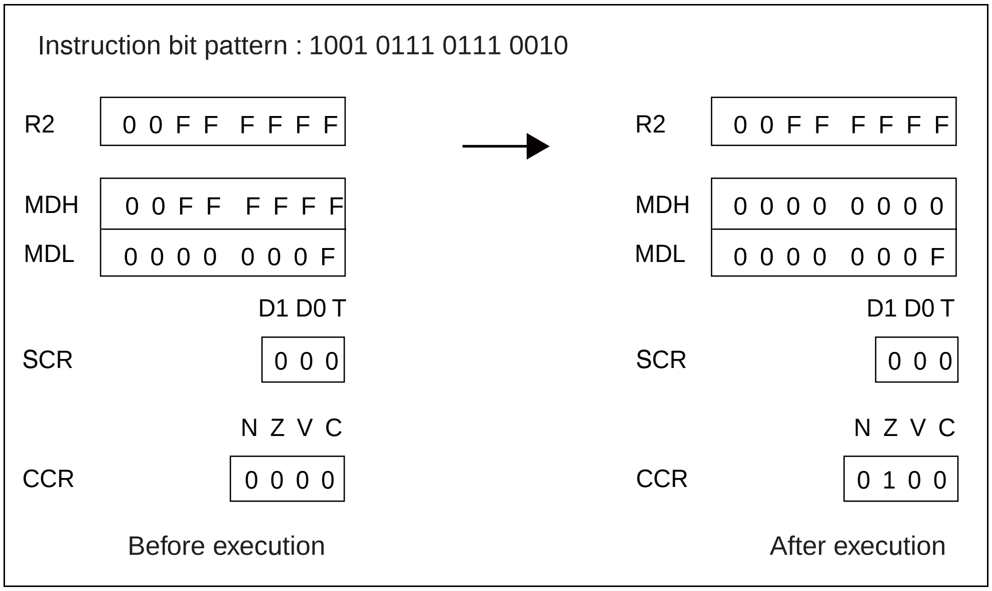
Execution cycles: 1 cycle

Instruction format:

MSB ... LSB

1	0	0	1	0	1	1	1	0	1	1	1	Ri

Example: DIV2 R2



7.42 DIV3 (Correction when Remainder is 0)

This instruction is used in signed division. It should be used in combinations such as DIV0S, DIV1 x 32, DIV2, DIV3 and DIV4S.

■ DIV3 (Correction when Remainder is 0)

Assembler format: DIV3

Operation:

```
if (Z = = 1) {
    MDL + 1 → MDL
}
```

Flag change:

N	Z	V	C
–	–	–	–

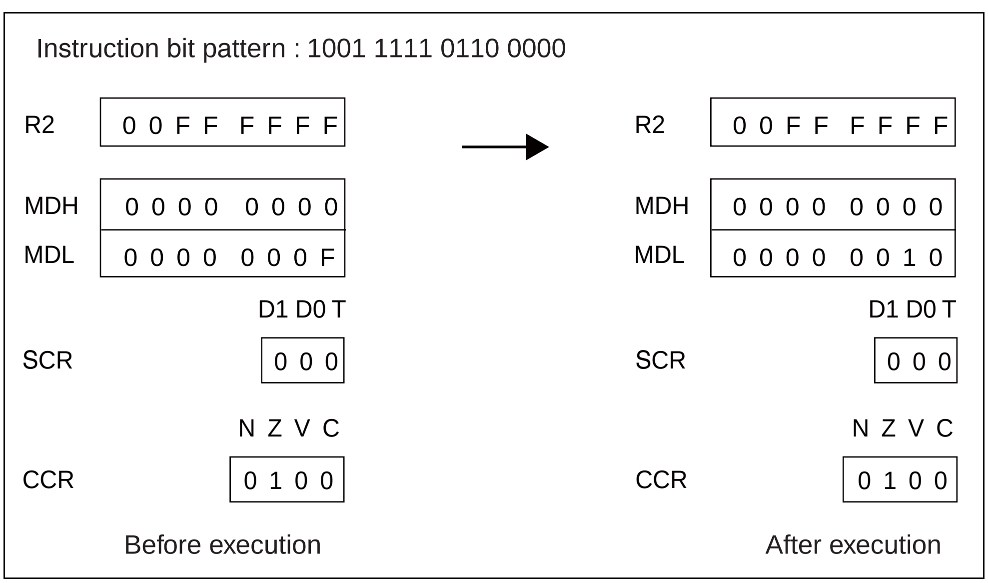
N, Z, V, and C: Unchanged

Execution cycles: 1 cycle

Instruction format:

MSB															LSB
1	0	0	1	1	1	1	1	0	1	1	0	0	0	0	0

Example: DIV3



7.43 DIV4S (Correction Answer for Signed Division)

This instruction is used in signed division. It should be used in combinations such as DIV0S, DIV1 x 32, DIV2, DIV3 and DIV4S.

■ DIV4S (Correction Answer for Signed Division)

Assembler format: DIV4S

Operation:
```
if (D1 = = 1) {
    0 – MDL → MDL
}
```

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

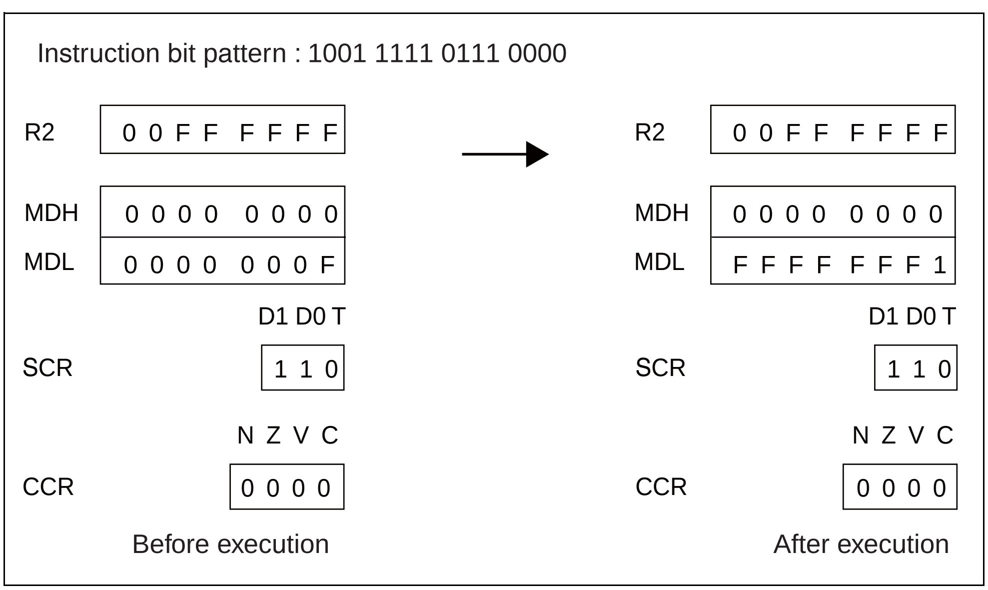
Execution cycles: 1 cycle

Instruction format:

MSB ... LSB

1	0	0	1	1	1	1	1	0	1	1	1	0	0	0	0

Example: DIV4S



7.44 LSL (Logical Shift to the Left Direction)

Makes a logical left shift of the word data in "Ri" by "Rj" bits, stores the result to "Ri". Only the lower 5 bits of "Rj", which designates the size of the shift, are valid and the shift range is 0 to 31 bits.

■ LSL (Logical Shift to the Left Direction)

Assembler format: LSL Rj, Ri

Operation: Ri << Rj → Ri

Flag change:

N	Z	V	C
C	C	–	C

N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is "0", cleared otherwise.

V: Unchanged

C: Holds the bit value shifted last. Cleared when the shift amount is "0".

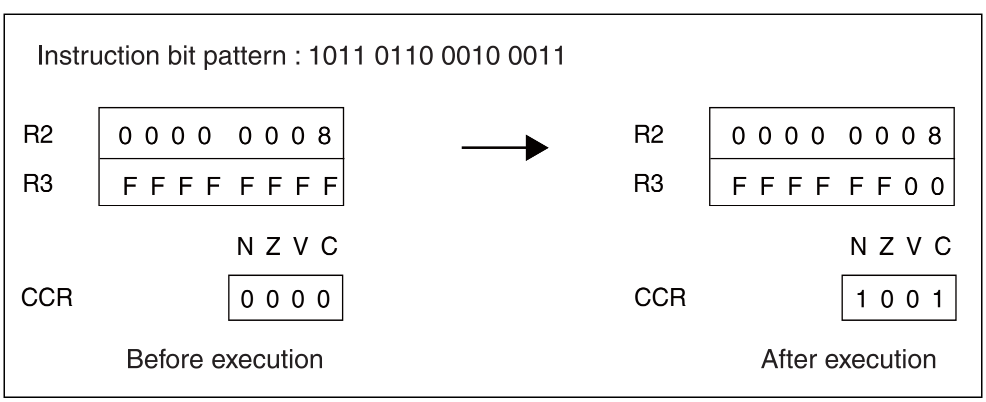
Execution cycles: 1 cycle

Instruction format:

MSB … LSB

1	0	1	1	0	1	1	0	Rj	Ri

Example: LSL R2, R3



7.45 LSL (Logical Shift to the Left Direction)

Makes a logical left shift of the word data in "Ri" by "u4" bits, stores the result to "Ri".

■ LSL (Logical Shift to the Left Direction)

Assembler format: LSL #u4, Ri

Operation: Ri << u4 → Ri

Flag change:

N	Z	V	C
C	C	–	C

N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is "0", cleared otherwise.

V: Unchanged

C: Holds the bit value shifted last. Cleared when the shift amount is "0".

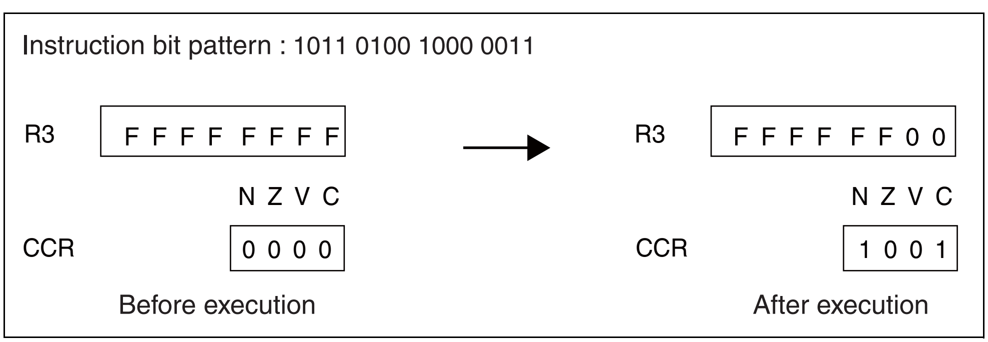
Execution cycles: 1 cycle

Instruction format:

MSB … LSB

1	0	1	1	0	1	0	0	u4	Ri

Example: LSL #8, R3



7.46 LSL2 (Logical Shift to the Left Direction)

Makes a logical left shift of the word data in "Ri" by "{u4 + 16}" bits, stores the results to "Ri".

■ LSL2 (Logical Shift to the Left Direction)

Assembler format: LSL2 #u4, Ri

Operation: Ri << {u4 + 16} → Ri

Flag change:

N	Z	V	C
C	C	–	C

N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is "0", cleared otherwise.

V: Unchanged

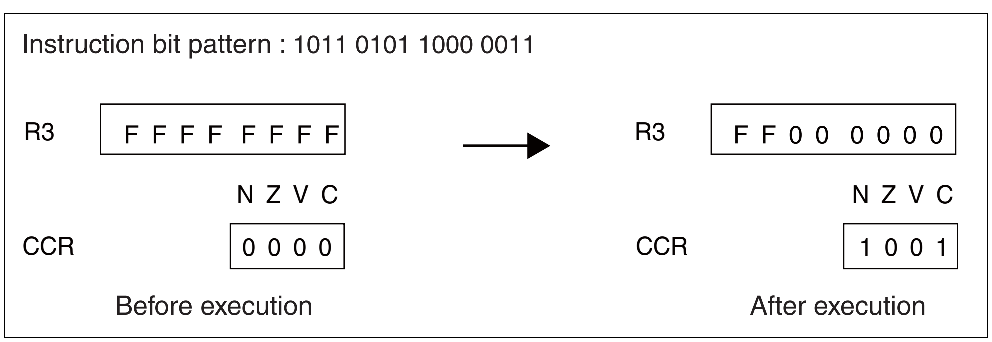
C: Holds the bit value shifted last.

Execution cycles: 1 cycle

Instruction format:

MSB									LSB
1	0	1	1	0	1	0	1	u4	Ri

Example: LSL2 #8, R3



7.47 LSR (Logical Shift to the Right Direction)

Makes a logical right shift of the word data in "Ri" by "Rj" bits, stores the result to "Ri". Only the lower 5 bits of "Rj", which designates the size of the shift, are valid and the shift range is 0 to 31 bits.

■ LSR (Logical Shift to the Right Direction)

Assembler format: LSR Rj, Ri

Operation: Ri >> Rj → Ri

Flag change:

N	Z	V	C
C	C	–	C

N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is "0", cleared otherwise.

V: Unchanged

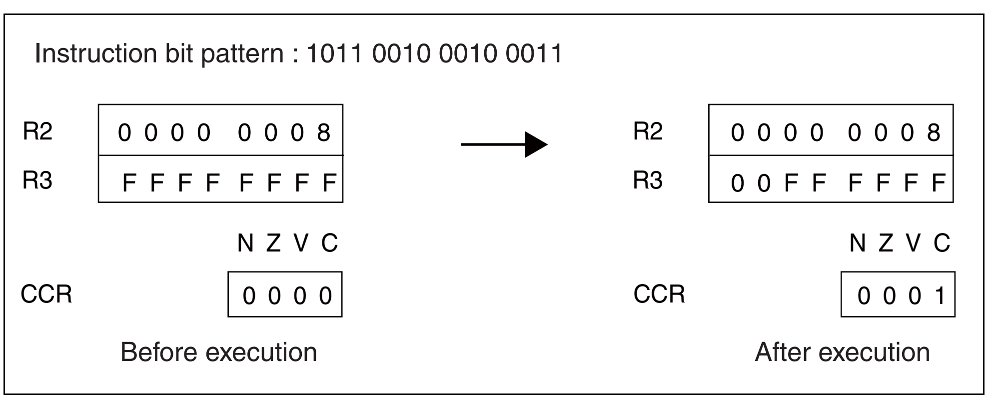
C: Holds the bit value shifted last. Cleared when the shift amount is "0".

Execution cycles: 1 cycle

Instruction format:

MSB									LSB
1	0	1	1	0	0	1	0	Rj	Ri

Example: LSR R2, R3



7.48 LSR (Logical Shift to the Right Direction)

Makes a logical right shift of the word data in "Ri" by "u4" bits, stores the result to "Ri".

■ LSR (Logical Shift to the Right Direction)

Assembler format: LSR #u4, Ri

Operation: Ri >> u4 → Ri

Flag change:

N	Z	V	C
C	C	–	C

N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is "0", cleared otherwise.

V: Unchanged

C: Holds the bit value shifted last. Cleared when the shift amount is "0".

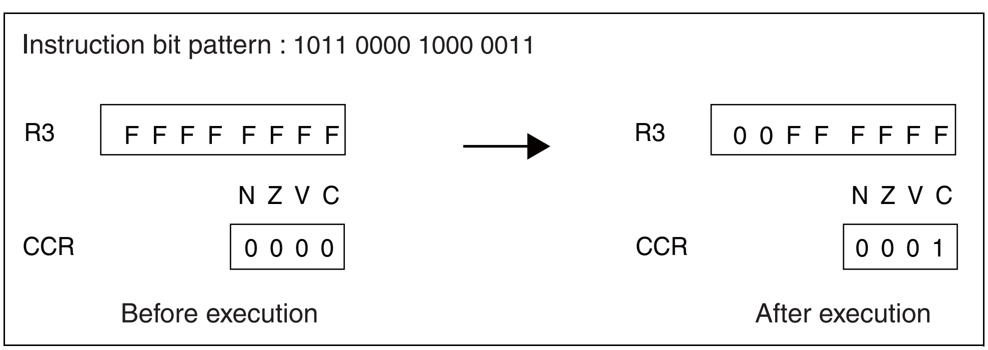
Execution cycles: 1 cycle

Instruction format:

MSB ... LSB

1	0	1	1	0	0	0	0	u4	Ri

Example: LSR #8, R3



7.49 LSR2 (Logical Shift to the Right Direction)

Makes a logical right shift of the word data in "Ri" by "{u4 + 16}" bits, stores the result to "Ri".

■ LSR2 (Logical Shift to the Right Direction)

Assembler format: LSR2 #u4, Ri

Operation: Ri >> {u4 + 16} → Ri

Flag change:

N	Z	V	C
0	C	–	C

N: Cleared

Z: Set when the operation result is "0", cleared otherwise.

V: Unchanged

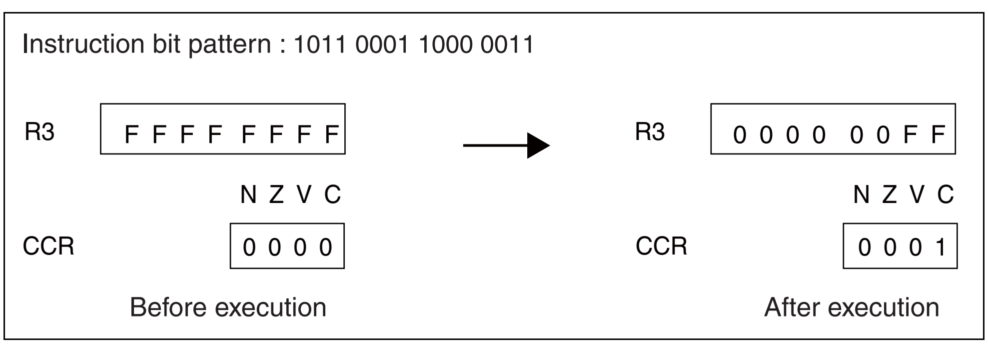
C: Holds the bit value shifted last.

Execution cycles: 1 cycle

Instruction format:

MSB									LSB
1	0	1	1	0	0	0	1	u4	Ri

Example: LSR2 #8, R3



7.50 ASR (Arithmetic Shift to the Right Direction)

Makes an arithmetic right shift of the word data in "Ri" by "Rj" bits, stores the result to "Ri".
Only the lower 5 bits of "Rj", which designates the size of the shift, are valid and the shift range is 0 to 31 bits.

■ ASR (Arithmetic Shift to the Right Direction)

Assembler format: ASR Rj, Ri

Operation: Ri >> Rj → Ri

Flag change:

N	Z	V	C
C	C	–	C

N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is "0", cleared otherwise.

V: Unchanged

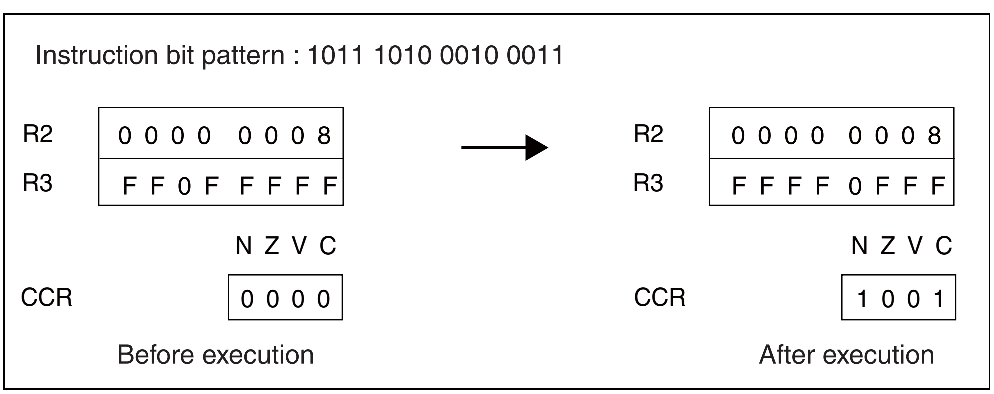
C: Holds the bit value shifted last. Cleared when the shift amount is "0".

Execution cycles: 1 cycle

Instruction format:

MSB									LSB
1	0	1	1	1	0	1	0	Rj	Ri

Example: ASR R2, R3



7.51 ASR (Arithmetic Shift to the Right Direction)

Makes an arithmetic right shift of the word data in "Ri" by "u4" bits, stores the result to "Ri".

■ ASR (Arithmetic Shift to the Right Direction)

Assembler format: ASR #u4, Ri

Operation: Ri >> u4 → Ri

Flag change:

N	Z	V	C
C	C	–	C

N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is "0", cleared otherwise.

V: Unchanged

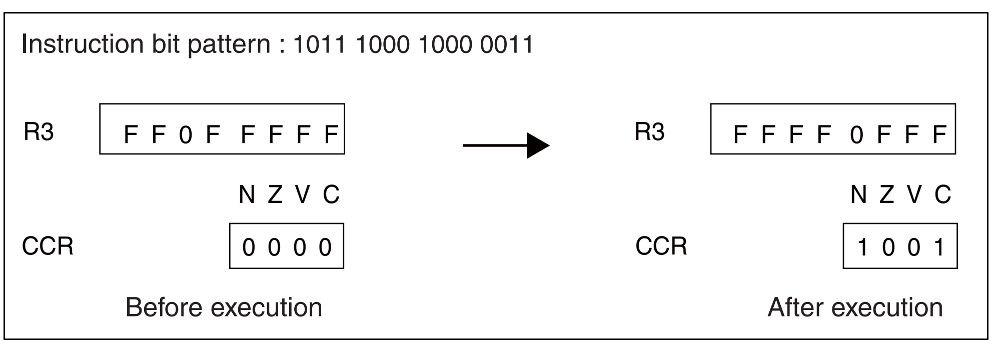
C: Holds the bit value shifted last. Cleared when the shift amount is "0".

Execution cycles: 1 cycle

Instruction format:

MSB									LSB
1	0	1	1	1	0	0	0	u4	Ri

Example: ASR #8, R3



7.52 ASR2 (Arithmetic Shift to the Right Direction)

Makes an arithmetic right shift of the word data in "Ri" by "{u4 + 16}" bits, stores the result to "Ri".

■ ASR2 (Arithmetic Shift to the Right Direction)

Assembler format: ASR2 #u4, Ri

Operation: Ri >> {u4 + 16} → Ri

Flag change:

N	Z	V	C
C	C	–	C

N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is "0", cleared otherwise.

V: Unchanged

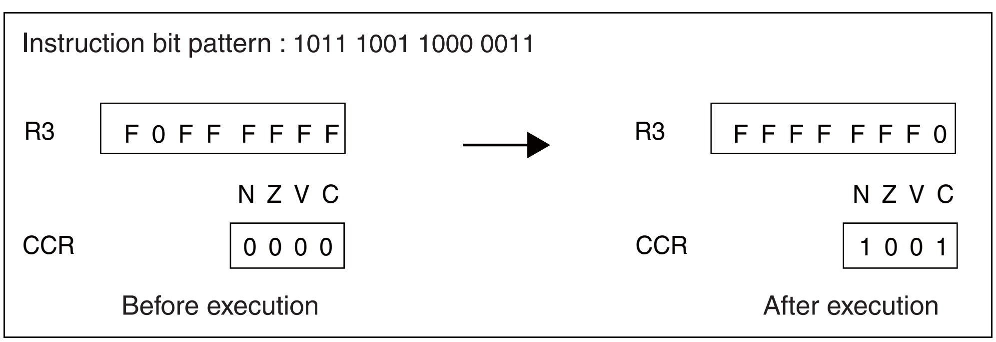
C: Holds the bit value shifted last.

Execution cycles: 1 cycle

Instruction format:

MSB									LSB
1	0	1	1	1	0	0	1	u4	Ri

Example: ASR2 #8, R3



7.53 LDI:32 (Load Immediate 32-bit Data to Destination Register)

Loads 1 word of immediate data to "Ri".

■ LDI:32 (Load Immediate 32-bit Data to Destination Register)

Assembler format: LDI:32 #i32, Ri

Operation: i32 → Ri

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

Execution cycles: 3 cycles

Instruction format:

	MSB												LSB
(n+0)	1	0	0	1	1	1	1	1	1	0	0	0	Ri
(n+2)	i32(higher)												
(n+4)	i32(lower)												

Example: LDI:32 #87654321H, R3

Instruction bit pattern : 1001 1111 1000 0011
: 1000 0111 0110 0101
: 0100 0011 0010 0001

	Before execution			After execution
R3	0 0 0 0 0 0 0 0	→	R3	8 7 6 5 4 3 2 1

7.54 LDI:20 (Load Immediate 20-bit Data to Destination Register)

Extends the 20-bit immediate data with 12 zeros in the higher bits, loads to "Ri".

■ LDI:20 (Load Immediate 20-bit Data to Destination Register)

Assembler format: LDI:20 #i20, Ri

Operation: extu (i20) → Ri

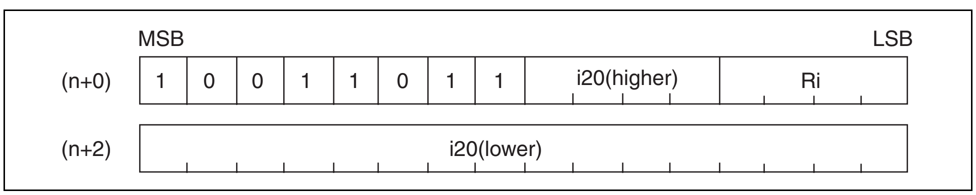
Flag change:

N	Z	V	C
–	–	–	–

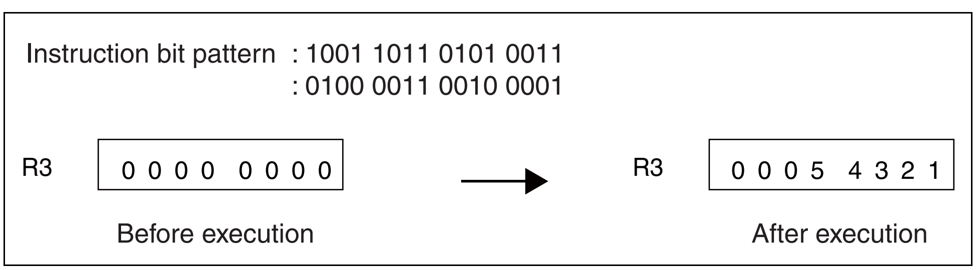
N, Z, V, and C: Unchanged

Execution cycles: 2 cycles

Instruction format:



Example: LDI:20 #54321H, R3



7.55 LDI:8 (Load Immediate 8-bit Data to Destination Register)

Extends the 8-bit immediate data with 24 zeros in the higher bits, loads to "Ri".

■ LDI:8 (Load Immediate 8-bit Data to Destination Register)

Assembler format: LDI:8 #i8, Ri

Operation: extu (i8) → Ri

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

Execution cycles: 1 cycle

Instruction format:

MSB ... LSB

1	1	0	0	i8	Ri

Example: LDI:8 #21H, R3

Instruction bit pattern : 1100 0010 0001 0011

	Before execution			After execution
R3	0 0 0 0 0 0 0 0	→	R3	0 0 0 0 0 0 2 1

7.56 LD (Load Word Data in Memory to Register)

Loads the word data at memory address "Rj" to "Ri".

■ LD (Load Word Data in Memory to Register)

Assembler format: LD @Rj, Ri

Operation: (Rj) → Ri

Flag change:

N	Z	V	C
–	–	–	–

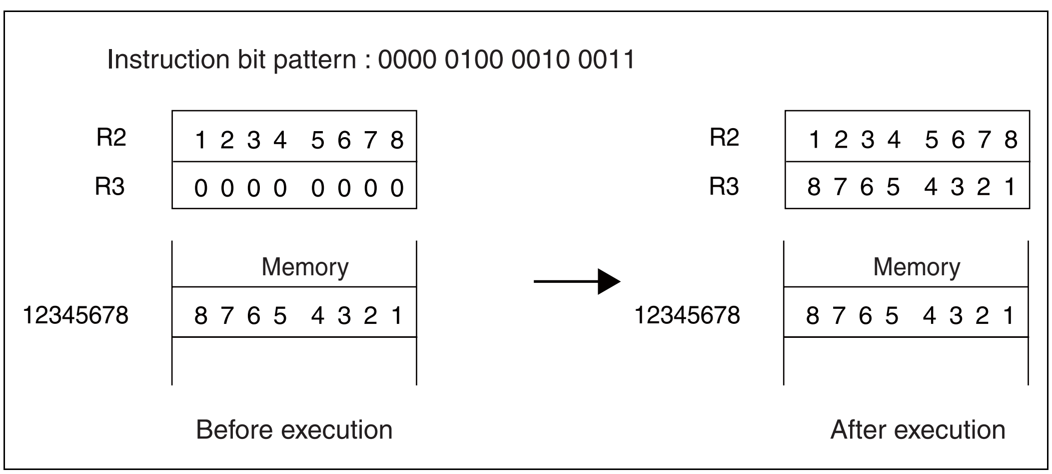
N, Z, V, and C: Unchanged

Execution cycles: b cycle(s)

Instruction format:

MSB									LSB
0	0	0	0	0	1	0	0	Rj	Ri

Example: LD @R2, R3



7.57 LD (Load Word Data in Memory to Register)

Loads the word data at memory address "(R13 + Rj)" to "Ri".

■ LD (Load Word Data in Memory to Register)

Assembler format: LD @ (R13, Rj), Ri

Operation: (R13 + Rj) → Ri

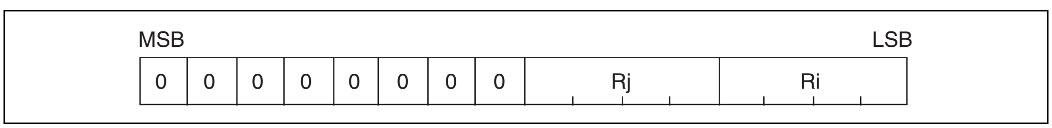
Flag change:

N	Z	V	C
–	–	–	–

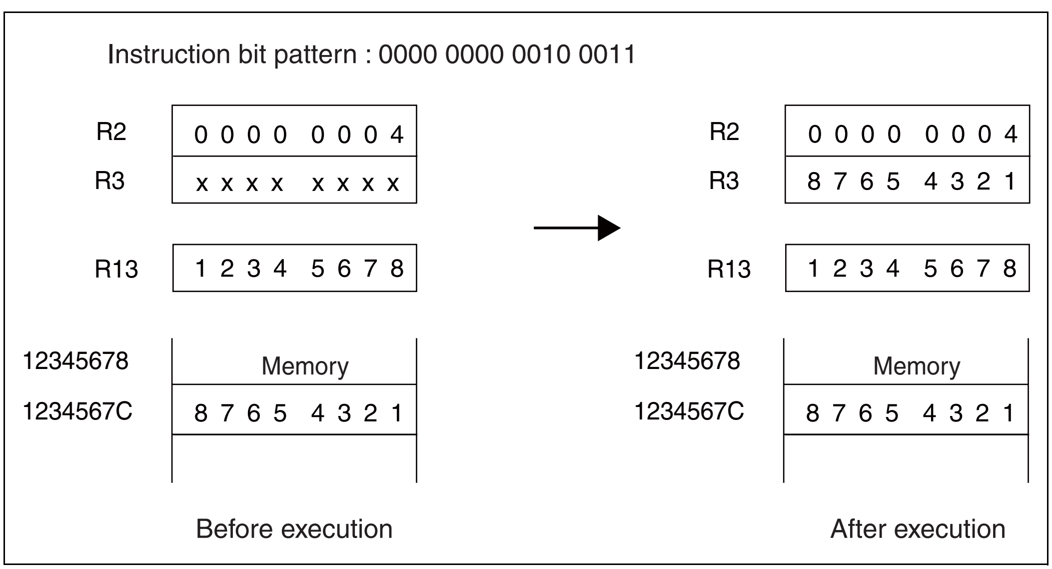
N, Z, V, and C: Unchanged

Execution cycles: b cycle(s)

Instruction format:



Example: LD @ (R13, R2), R3



7.58 LD (Load Word Data in Memory to Register)

Loads the word data at memory address "(R14 + o8 × 4)" to "Ri".
The value "o8" is a signed calculation.

■ LD (Load Word Data in Memory to Register)

Assembler format: LD @ (R14, disp10), Ri

Operation: (R14 + o8 × 4) → Ri

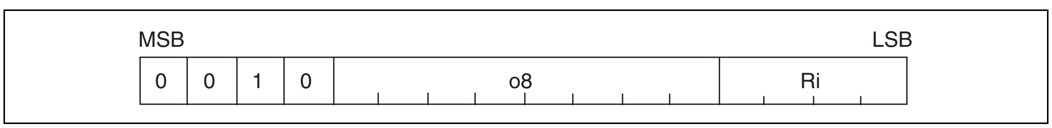
Flag change:

N	Z	V	C
–	–	–	–

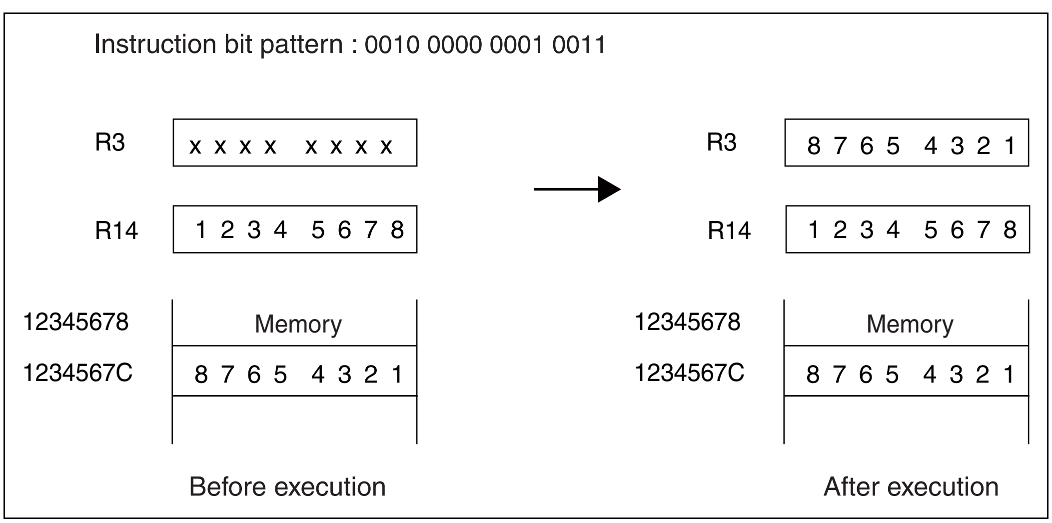
N, Z, V, and C: Unchanged

Execution cycles: b cycle(s)

Instruction format:



Example: LD @ (R14, 4), R3



7.59 LD (Load Word Data in Memory to Register)

Loads the word data at memory address "(R15 + u4 × 4)" to "Ri".
The value "u4" is an unsigned calculation.

■ LD (Load Word Data in Memory to Register)

Assembler format: LD @ (R15, udisp6), Ri

Operation: (R15 + u4 × 4) → Ri

Flag change:

N	Z	V	C
–	–	–	–

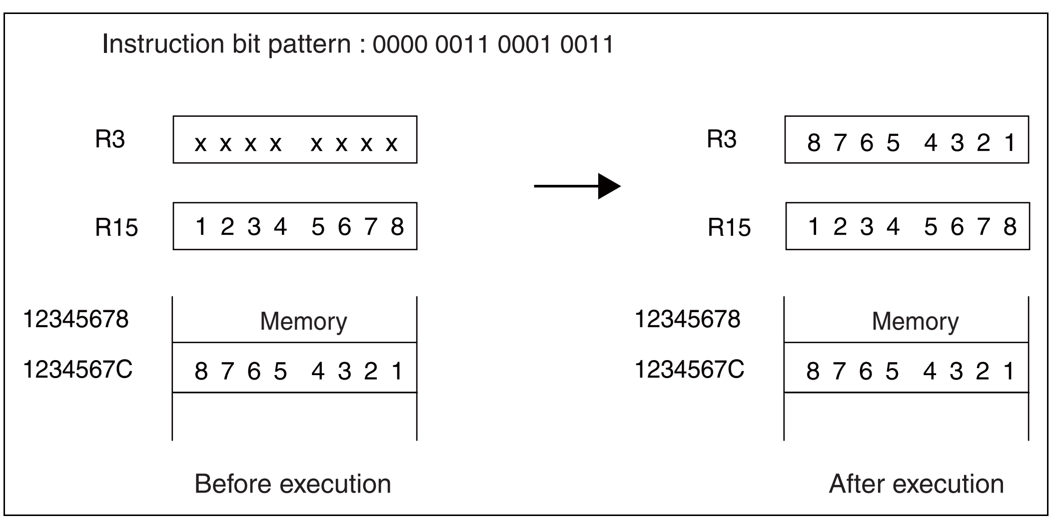
N, Z, V, and C: Unchanged

Execution cycles: b cycle(s)

Instruction format:

MSB									LSB
0	0	0	0	0	0	1	1	u4	Ri

Example: LD @ (R15, 4), R3



7.60 LD (Load Word Data in Memory to Register)

Loads the word data at memory address "R15" to "Rj", and adds 4 to the value of "R15". If "R15" is given as parameter "Ri", the value read from the memory will be loaded into memory address "R15".

■ LD (Load Word Data in Memory to Register)

Assembler format: LD @ R15 +, Ri

Operation: (R15) → Ri
R15 + 4 → R15

Flag change:

N	Z	V	C
–	–	–	–

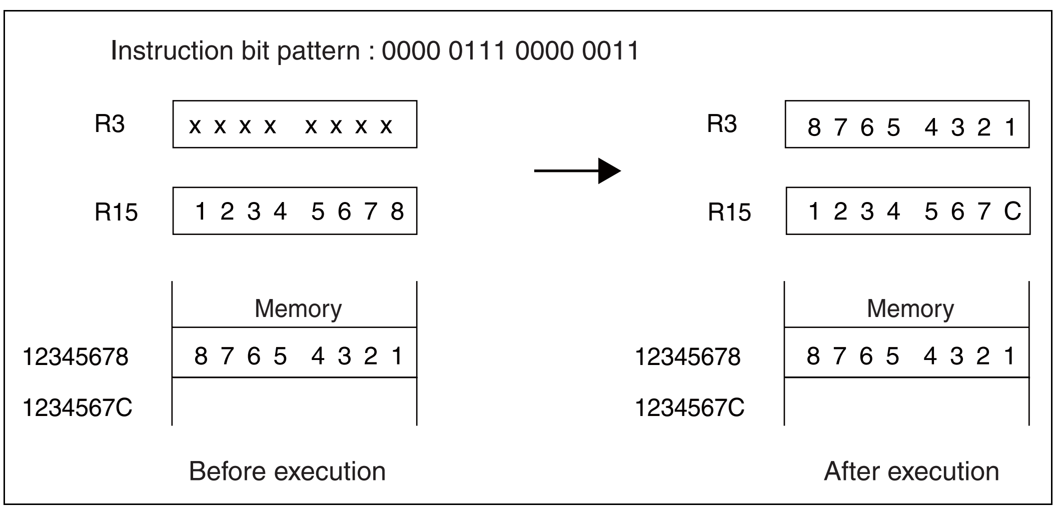
N, Z, V, and C: Unchanged

Execution cycles: b cycle(s)

Instruction format:

MSB												LSB
0	0	0	0	0	1	1	1	0	0	0	0	Ri

Example: LD @ R15 +, R3



7.61 LD (Load Word Data in Memory to Register)

Loads the word data at memory address "R15" to dedicated register "Rs", and adds 4 to the value of "R15".
If the number of a non-existent register is given as parameter "Rs", the read value "Ri" will be ignored.
If "Rs" is designated as the system stack pointer (SSP) or user stack pointer (USP), and that pointer is indicating "R15" [the "S" flag in the condition code register (CCR) is set to "0" to indicate the "SSP", and to "1" to indicate the "USP"], the last value remaining in "R15" will be the value read from memory.

■ LD (Load Word Data in Memory to Register)

Assembler format: LD @ R15 +, Rs

Operation: (R15) → Rs
R15 + 4 → R15

Flag change:

N	Z	V	C
–	–	–	–

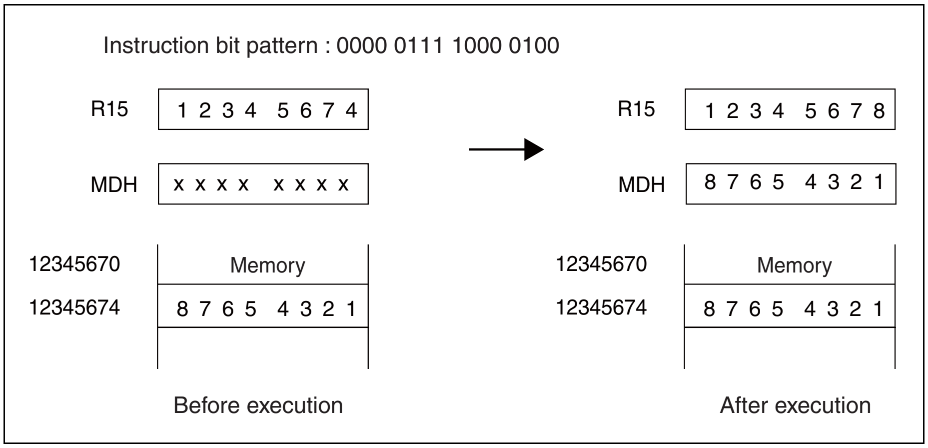
N, Z, V, and C: Unchanged

Execution cycles: b cycle(s)

Instruction format:

MSB												LSB
0	0	0	0	0	1	1	1	1	0	0	0	Rs

Example: LD @ R15 +, MDH



7.62 LD (Load Word Data in Memory to Program Status Register)

Loads the word data at memory address "R15" to the program status (PS), and adds 4 to the value of "R15".
At the time this instruction is executed, if the value of the interrupt level mask register (ILM) is in the range 16 to 31, only new "ILM" settings between 16 and 31 can be entered. If data in the range 0 to 15 is loaded from memory, the value 16 will be added to that data before being transferred to the "ILM". If the original "ILM" value is in the range 0 to 15, then any value from 0 to 31 can be transferred to the "ILM".

■ LD (Load Word Data in Memory to Program Status Register)

Assembler format: LD @ R15 +, PS

Operation: (R15) → PS
R15 + 4 → R15

Flag change:

N	Z	V	C
C	C	C	C

N, Z, V, and C: Data is transferred from "R15".

Execution cycles: 1 + a + c cycles

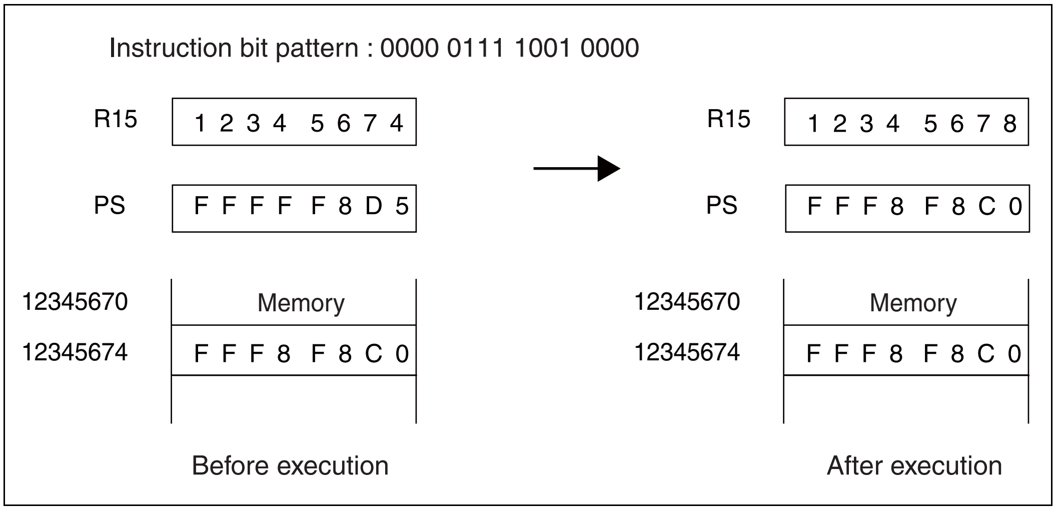
The value of "c" is normally 1 cycle. However, if the next instruction involves read or write access to memory address "R15", the system stack pointer (SSP) or the user stack pointer (USP), then an interlock is applied and the value becomes 2 cycles.

Instruction format:

MSB ... LSB

0	0	0	0	0	1	1	1	1	0	0	1	0	0	0	0

Example: LD @ R15 +, PS



7.63 LDUH (Load Half-word Data in Memory to Register)

Extends with zeros the half-word data at memory address "Rj", loads to "Ri".

■ LDUH (Load Half-word Data in Memory to Register)

Assembler format: LDUH @Rj, Ri

Operation: extu ((Rj)) → Ri

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

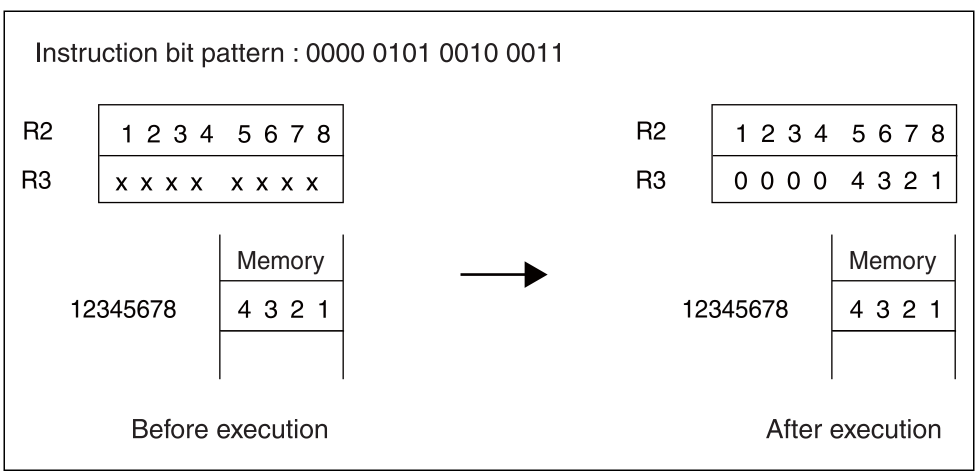
Execution cycles: b cycle(s)

Instruction format:

MSB ... LSB

0	0	0	0	0	1	0	1	Rj	Ri

Example: LDUH @R2, R3



7.64 LDUH (Load Half-word Data in Memory to Register)

Extends with zeros the half-word data at memory address "(R13 + Rj)", loads to "Ri".

■ LDUH (Load Half-word Data in Memory to Register)

Assembler format: LDUH @(R13, Rj), Ri

Operation: extu ((R13 + Rj)) → Ri

Flag change:

N	Z	V	C
–	–	–	–

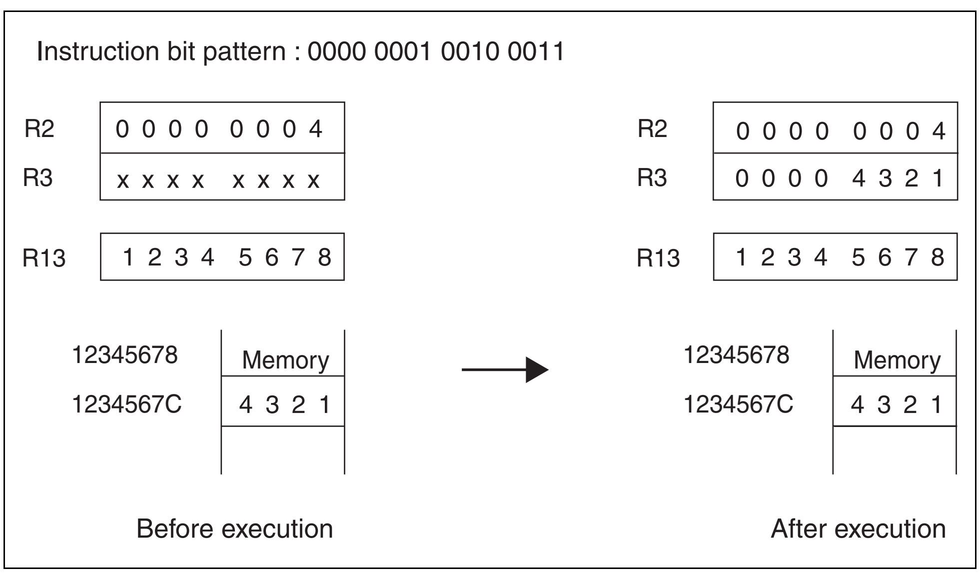
N, Z, V, and C: Unchanged

Execution cycles: b cycle(s)

Instruction format:

MSB									LSB
0	0	0	0	0	0	0	1	Rj	Ri

Example: LDUH @(R13, R2), R3



7.65 LDUH (Load Half-word Data in Memory to Register)

Extends with zeros the half-word data at memory address "(R14 + o8 × 2)", loads to "Ri".
The value "o8" is a signed calculation.

■ LDUH (Load Half-word Data in Memory to Register)

Assembler format: LDUH @(R14, disp9), Ri

Operation: extu ((R14 + o8 × 2)) → Ri

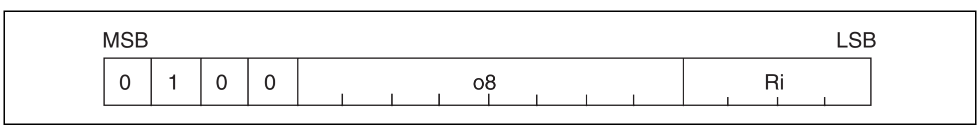
Flag change:

N	Z	V	C
–	–	–	–

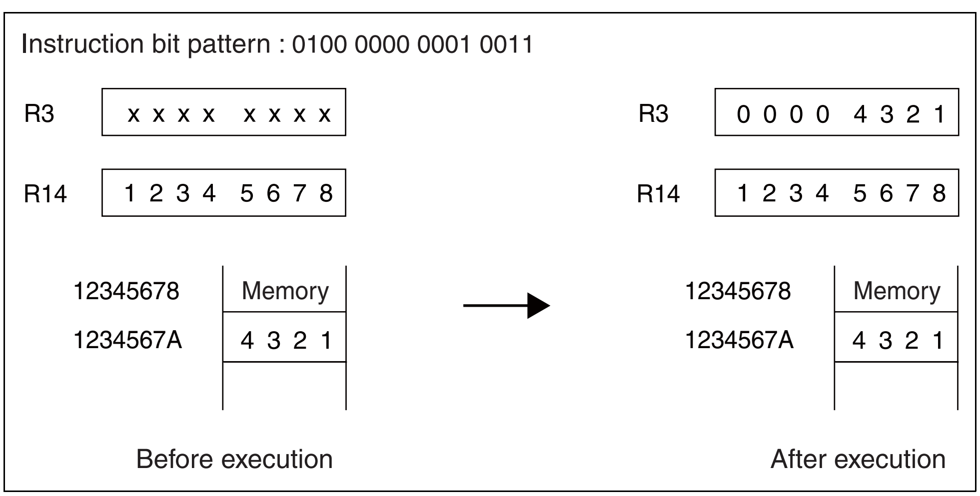
N, Z, V, and C: Unchanged

Execution cycles: b cycle(s)

Instruction format:



Example: LDUH @(R14, 2), R3



7.66 LDUB (Load Byte Data in Memory to Register)

Extends with zeros the byte data at memory address "Rj", loads to "Ri".

■ LDUB (Load Byte Data in Memory to Register)

Assembler format: LDUB @Rj, Ri

Operation: extu ((Rj)) → Ri

Flag change:

N	Z	V	C
–	–	–	–

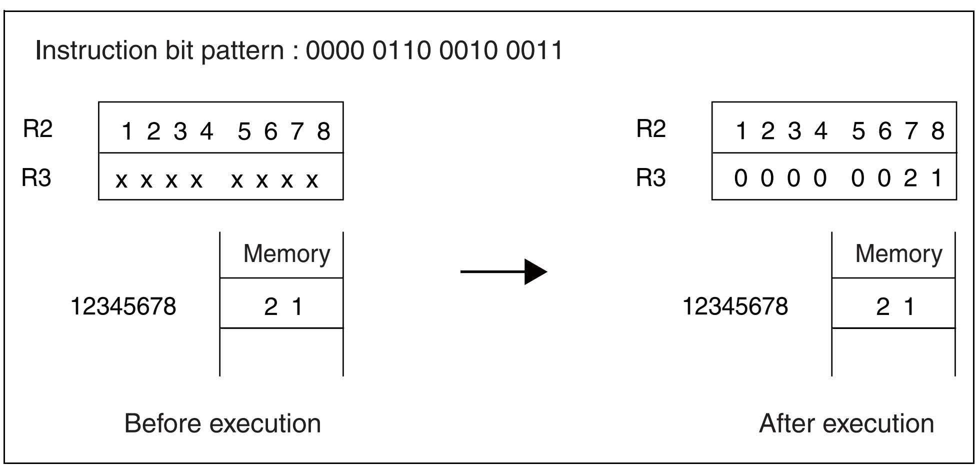
N, Z, V, and C: Unchanged

Execution cycles: b cycle(s)

Instruction format:

MSB									LSB
0	0	0	0	0	1	1	0	Rj	Ri

Example: LDUB @R2, R3



7.67 LDUB (Load Byte Data in Memory to Register)

Extends with zeros the byte data at memory address "(R13 + Rj)", loads to "Ri".

■ LDUB (Load Byte Data in Memory to Register)

Assembler format: LDUB @ (R13, Rj), Ri

Operation: extu ((R13 + Rj)) → Ri

Flag change:

N	Z	V	C
–	–	–	–

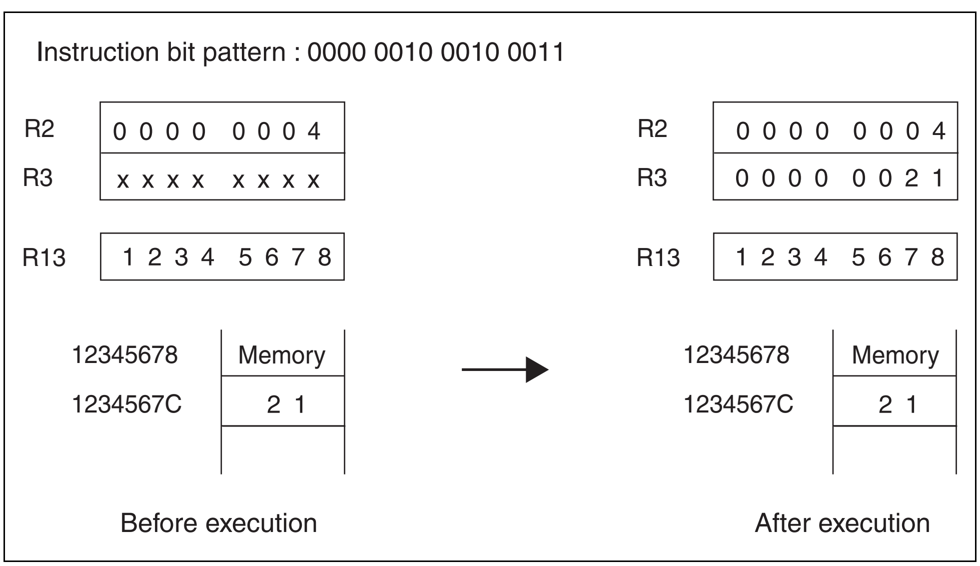
N, Z, V, and C: Unchanged

Execution cycles: b cycle(s)

Instruction format:

MSB									LSB
0	0	0	0	0	0	1	0	Rj	Ri

Example: LDUB @(R13, R2), R3



7.68 LDUB (Load Byte Data in Memory to Register)

Extends with zeros the byte data at memory address "(R14 + o8)", loads to "Ri". The value "o8" is a signed calculation.

■ LDUB (Load Byte Data in Memory to Register)

Assembler format: LDUB @ (R14, disp8), Ri

Operation: extu ((R14 + o8)) → Ri

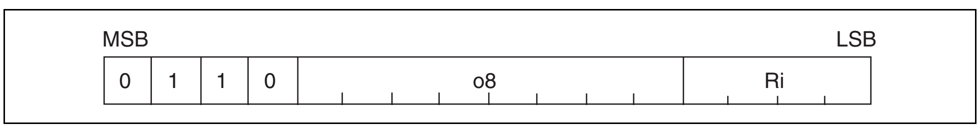
Flag change:

N	Z	V	C
–	–	–	–

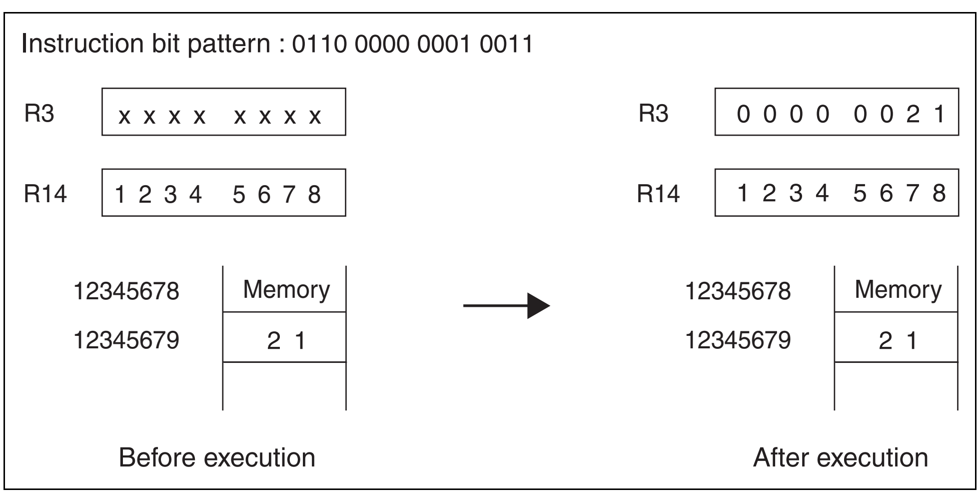
N, Z, V, and C: Unchanged

Execution cycles: b cycle(s)

Instruction format:



Example: LDUB @(R14, 1), R3



7.69 ST (Store Word Data in Register to Memory)

Loads the word data in "Ri" to memory address "Rj".

■ ST (Store Word Data in Register to Memory)

Assembler format: ST Ri, @Rj

Operation: Ri → (Rj)

Flag change:

N	Z	V	C
–	–	–	–

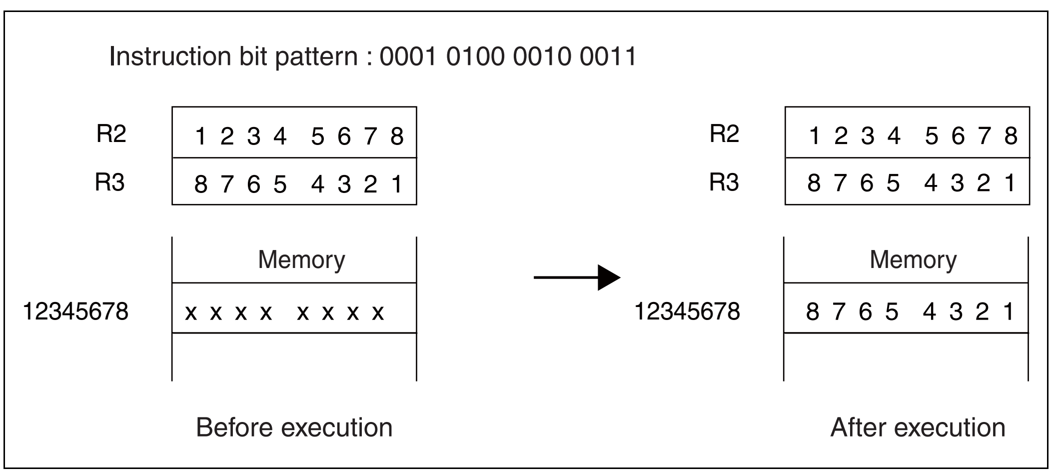
N, Z, V, and C: Unchanged

Execution cycles: a cycle(s)

Instruction format:

MSB									LSB
0	0	0	1	0	1	0	0	Rj	Ri

Example: ST R3, @R2



7.70 ST (Store Word Data in Register to Memory)

Loads the word data in "Ri" to memory address "(R13 + Rj)".

■ ST (Store Word Data in Register to Memory)

Assembler format: ST Ri, @ (R13, Rj)

Operation: Ri → (R13 + Rj)

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

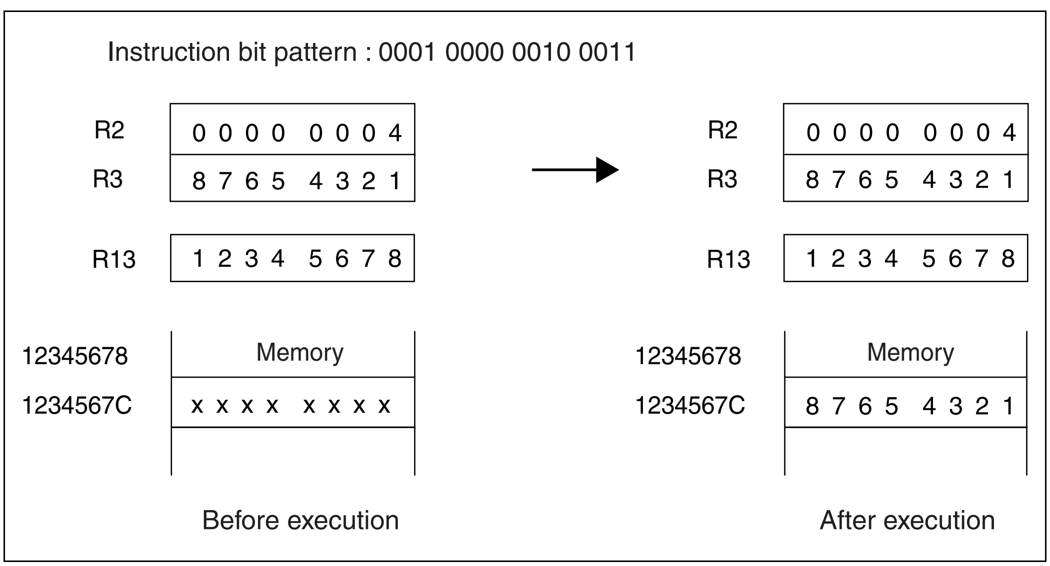
Execution cycles: a cycle(s)

Instruction format:

MSB … LSB

0	0	0	1	0	0	0	0	Rj	Ri

Example: ST R3, @ (R13, R2)



7.71 ST (Store Word Data in Register to Memory)

Loads the word data in "Ri" to memory address "(R14 + o8 × 4)".
The value "o8" is a signed calculation.

■ ST (Store Word Data in Register to Memory)

Assembler format: ST Ri,@ (R14, disp10)

Operation: Ri → (R14 + o8 × 4)

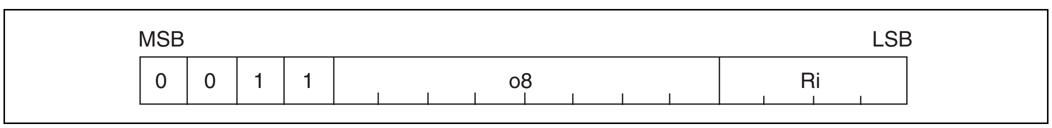
Flag change:

N	Z	V	C
–	–	–	–

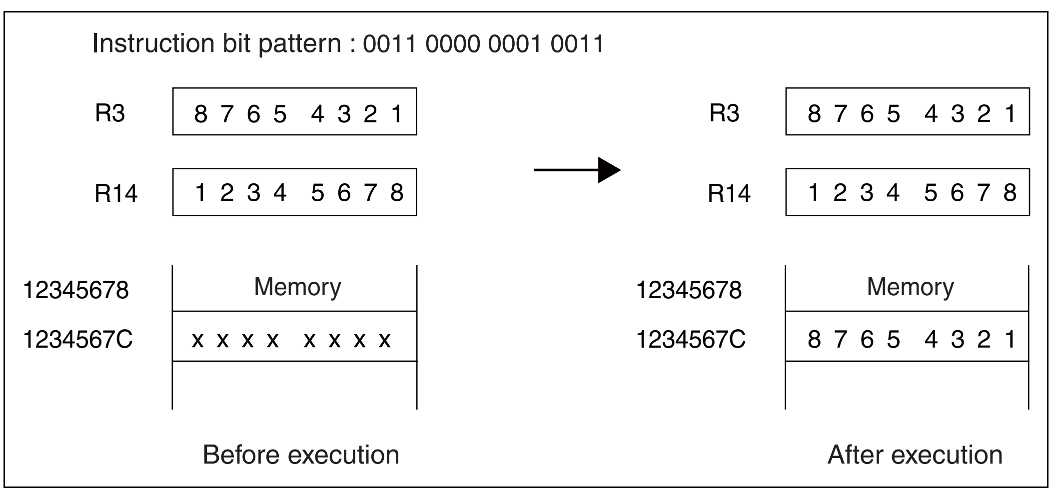
N, Z, V, and C: Unchanged

Execution cycles: a cycle(s)

Instruction format:



Example: ST R3, @ (R14, 4)



7.72 ST (Store Word Data in Register to Memory)

Loads the word data in "Ri" to memory address "(R15 + u4 × 4)".
The value "u4" is an unsigned calculation.

■ ST (Store Word Data in Register to Memory)

Assembler format: ST Ri, @ (R15, udisp6)

Operation: Ri → (R15 + u4 × 4)

Flag change:

N	Z	V	C
–	–	–	–

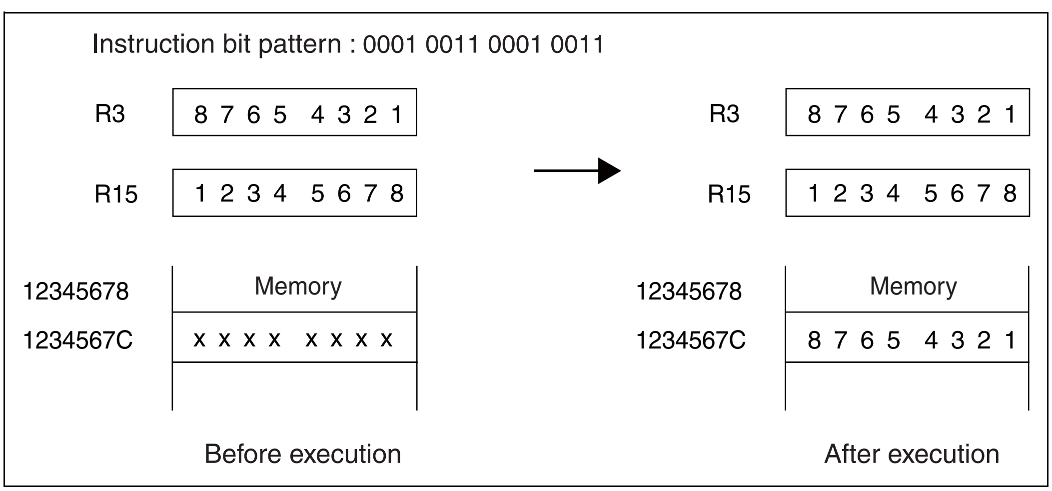
N, Z, V, and C: Unchanged

Execution cycles: a cycle(s)

Instruction format:

MSB									LSB
0	0	0	1	0	0	1	1	u4	Ri

Example: ST R3, @ (R15, 4)



7.73 ST (Store Word Data in Register to Memory)

Subtracts 4 from the value of "R15", stores the word data in "Ri" to the memory address indicated by the new value of "R15".
If "R15" is given as the parameter "Ri", the data transfer will use the value of "R15" before subtraction.

■ ST (Store Word Data in Register to Memory)

Assembler format: ST Ri, @ – R15

Operation: R15 – 4 → R15
Ri → (R15)

Flag change:

N	Z	V	C
–	–	–	–

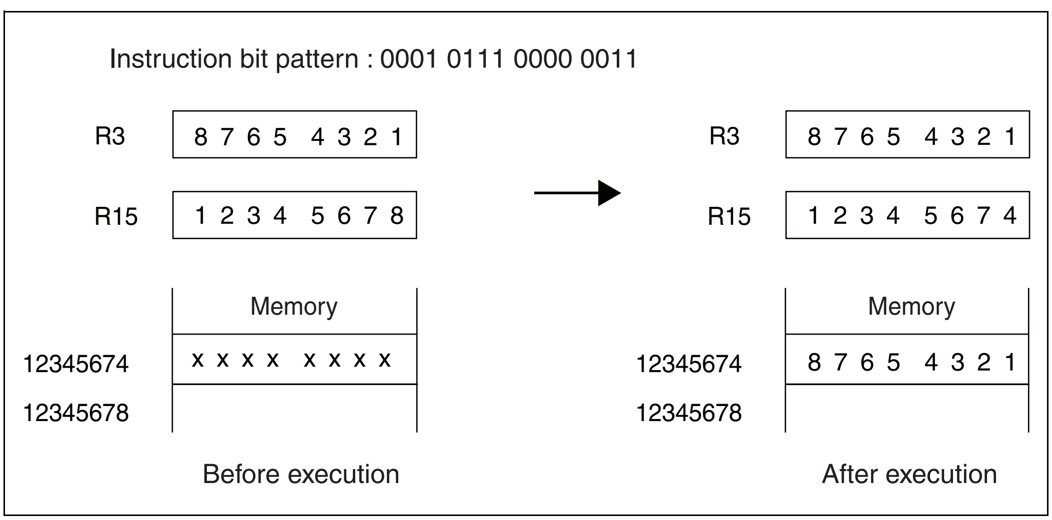
N, Z, V, and C: Unchanged

Execution cycles: a cycle(s)

Instruction format:

MSB												LSB
0	0	0	1	0	1	1	1	0	0	0	0	Ri

Example: ST R3, @ – R15



7.74 ST (Store Word Data in Register to Memory)

Subtracts 4 from the value of "R15", stores the word data in dedicated register "Rs" to the memory address indicated by the new value of "R15".
If a non-existent dedicated register is given as "Rs", undefined data will be transferred.

■ ST (Store Word Data in Register to Memory)

Assembler format: ST Rs, @ – R15

Operation: R15 – 4 → R15
Rs → (R15)

Flag change:

N	Z	V	C
–	–	–	–

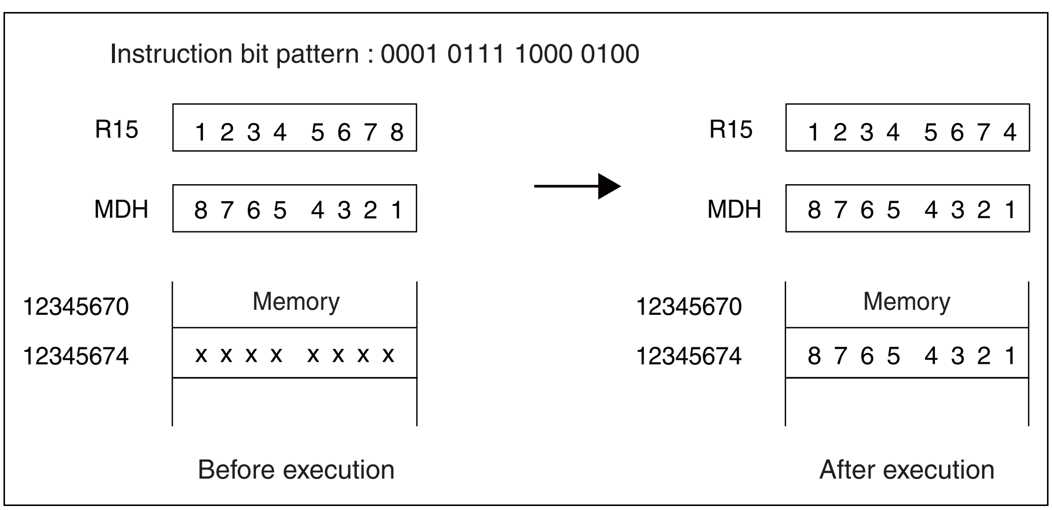
N, Z, V, and C: Unchanged

Execution cycles: a cycle(s)

Instruction format:

MSB												LSB
0	0	0	1	0	1	1	1	1	0	0	0	Rs

Example: ST MDH, @ – R15



7.75 ST (Store Word Data in Program Status Register to Memory)

Subtracts 4 from the value of "R15", stores the word data in the program status (PS) to the memory address indicated by the new value of "R15".

■ ST (Store Word Data in Program Status Register to Memory)

Assembler format: ST PS, @ – R15

Operation: R15 – 4 → R15
PS → (R15)

Flag change:

N	Z	V	C
–	–	–	–

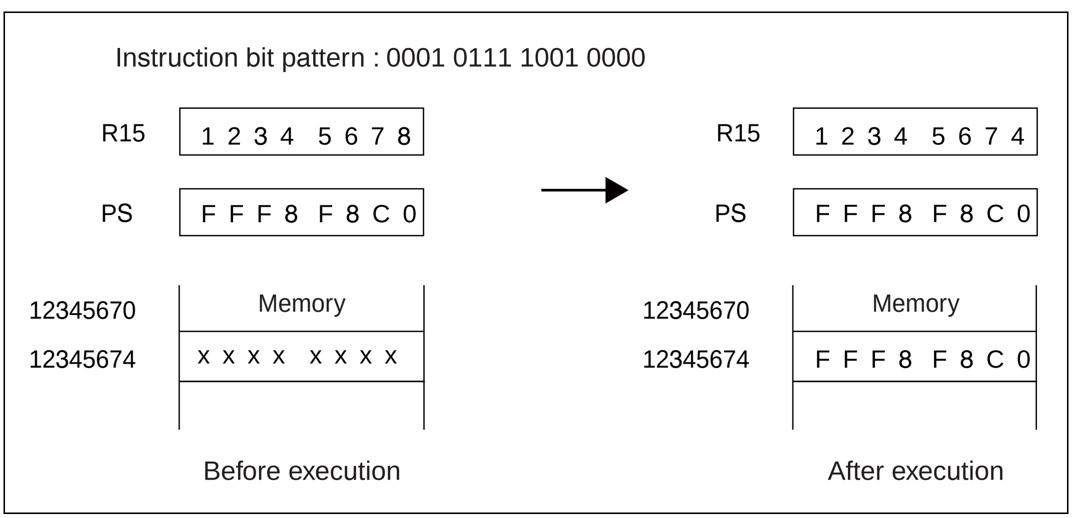
N, Z, V, and C: Unchanged

Execution cycles: a cycle(s)

Instruction format:

MSB															LSB
0	0	0	1	0	1	1	1	1	0	0	1	0	0	0	0

Example: ST PS, @ – R15



7.76 STH (Store Half-word Data in Register to Memory)

Stores the half-word data in "Ri" to memory address "Rj".

■ STH (Store Half-word Data in Register to Memory)

Assembler format: STH Ri, @Rj

Operation: Ri → (Rj)

Flag change:

N	Z	V	C
–	–	–	–

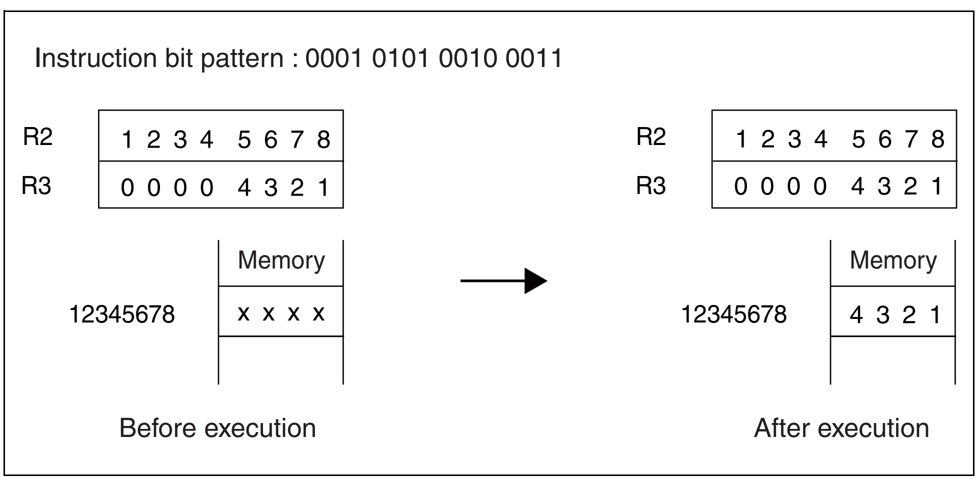
N, Z, V, and C: Unchanged

Execution cycles: a cycle(s)

Instruction format:

MSB									LSB
0	0	0	1	0	1	0	1	Rj	Ri

Example: STH R3, @R2



7.77 STH (Store Half-word Data in Register to Memory)

Stores the half-word data in "Ri" to memory address "(R13 + Rj)".

■ STH (Store Half-word Data in Register to Memory)

Assembler format: STH Ri, @(R13, Rj)

Operation: Ri → (R13 + Rj)

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

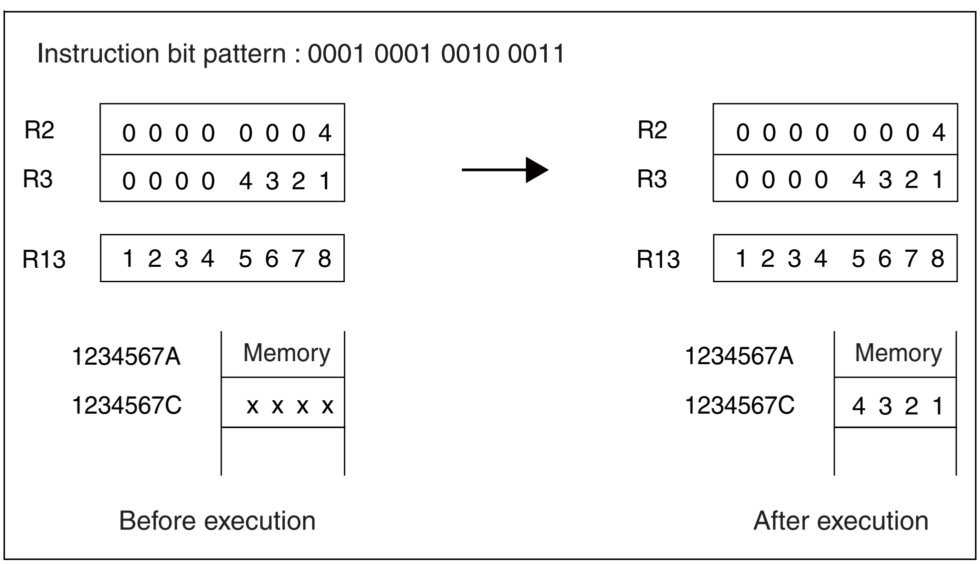
Execution cycles: a cycle(s)

Instruction format:

MSB ... LSB

0	0	0	1	0	0	0	1	Rj	Ri

Example: STH R3, @(R13, R2)



7.78 STH (Store Half-word Data in Register to Memory)

Stores the half-word data in "Ri" to memory address "(R14 + o8 × 2)".
The value "o8" is a signed calculation.

■ STH (Store Half-word Data in Register to Memory)

Assembler format: STH Ri, @(R14, disp9)

Operation: Ri → (R14 + o8 × 2)

Flag change:

N	Z	V	C
–	–	–	–

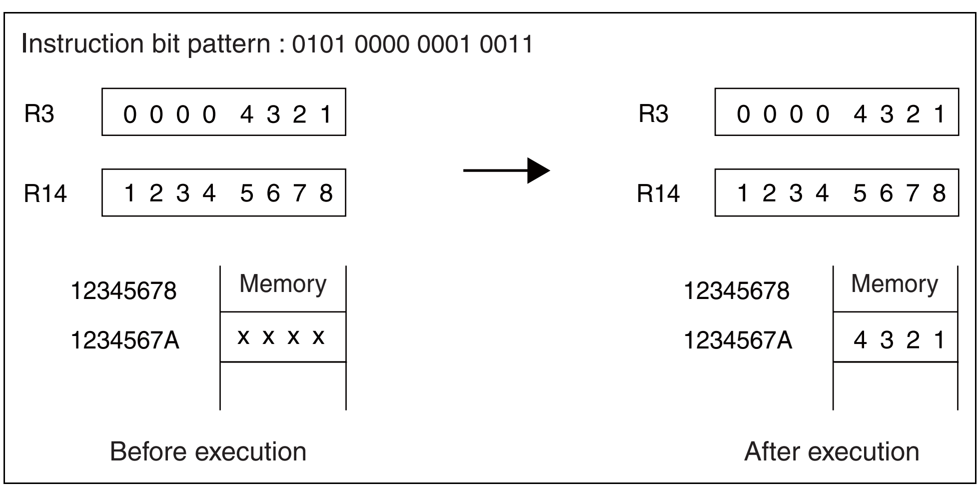
N, Z, V, and C: Unchanged

Execution cycles: a cycle(s)

Instruction format:

MSB					LSB
0	1	0	1	o8	Ri

Example: STH R3, @(R14, 2)



7.79 STB (Store Byte Data in Register to Memory)

Stores the byte data in "Ri" to memory address "Rj".

■ STB (Store Byte Data in Register to Memory)

Assembler format: STB Ri, @Rj

Operation: Ri → (Rj)

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

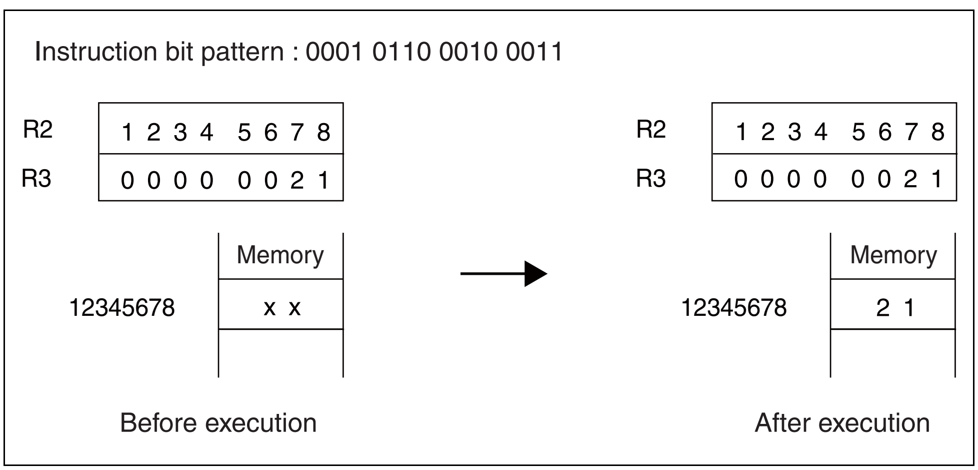
Execution cycles: a cycle(s)

Instruction format:

MSB ... LSB

0	0	0	1	0	1	1	0	Rj	Ri

Example: STB R3, @R2



7.80 STB (Store Byte Data in Register to Memory)

Stores the byte data in "Ri" to memory address "(R13 + Rj)".

■ STB (Store Byte Data in Register to Memory)

Assembler format: STB Ri, @ (R13, Rj)

Operation: Ri → (R13 + Rj)

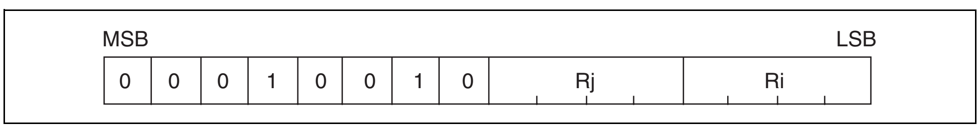
Flag change:

N	Z	V	C
–	–	–	–

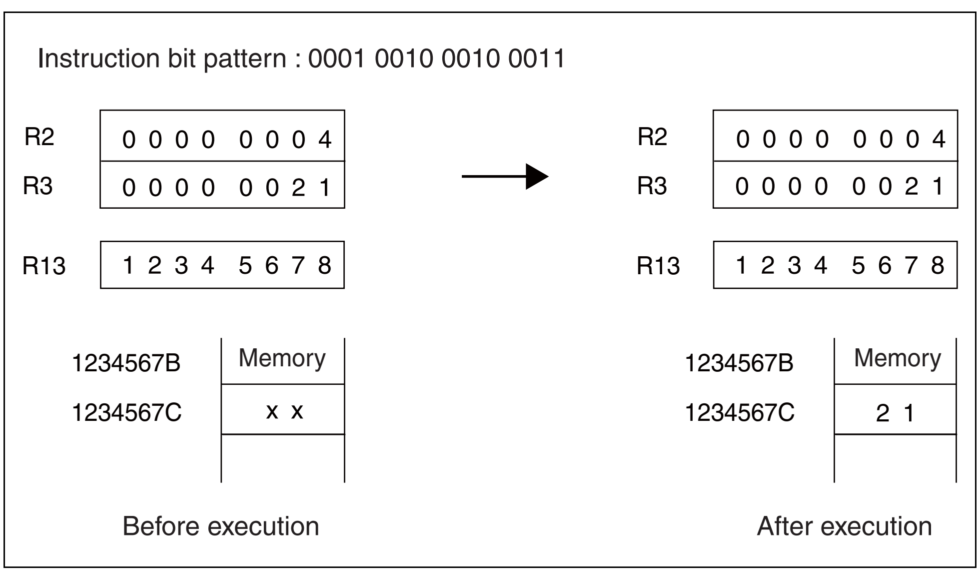
N, Z, V, and C: Unchanged

Execution cycles: a cycle(s)

Instruction format:



Example: STB R3, @(R13, R2)



7.81 STB (Store Byte Data in Register to Memory)

Stores the byte data in "Ri" to memory address "(R14 + o8)".
The value "o8" is a signed calculation.

■ STB (Store Byte Data in Register to Memory)

Assembler format: STB Ri, @ (R14, disp8)

Operation: Ri → (R14 + o8)

Flag change:

N	Z	V	C
–	–	–	–

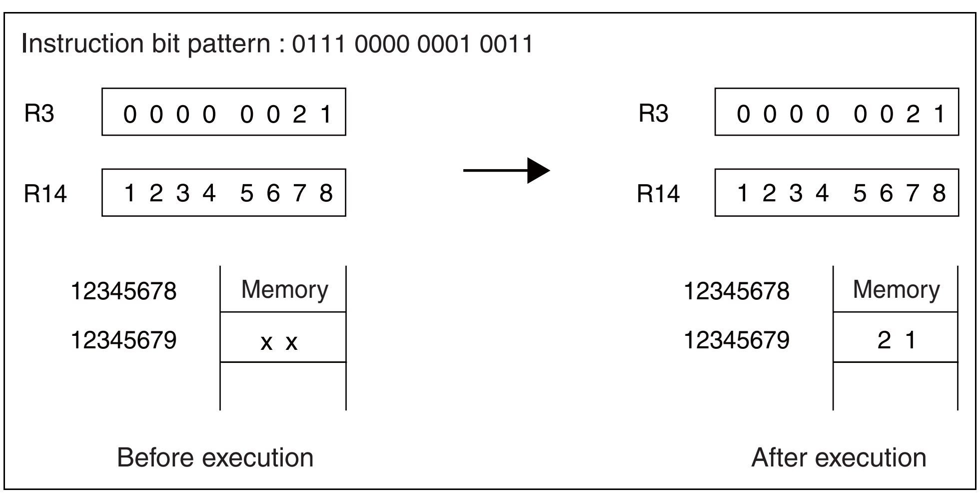
N, Z, V, and C: Unchanged

Execution cycles: a cycle(s)

Instruction format:

MSB					LSB
0	1	1	1	o8	Ri

Example: STB R3, @(R14, 1)



7.82 MOV (Move Word Data in Source Register to Destination Register)

Moves the word data in "Rj" to "Ri".

■ MOV (Move Word Data in Source Register to Destination Register)

Assembler format: MOV Rj, Ri

Operation: Rj → Ri

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

Execution cycles: 1 cycle

Instruction format:

MSB ... LSB

1	0	0	0	1	0	1	1	Rj	Ri

Example: MOV R2, R3

Instruction bit pattern : 1000 1011 0010 0011

	Before execution		After execution
R2	8 7 6 5 4 3 2 1	→	8 7 6 5 4 3 2 1
R3	x x x x x x x x		8 7 6 5 4 3 2 1

7.83 MOV (Move Word Data in Source Register to Destination Register)

Moves the word data in dedicated register "Rs" to general-purpose register "Ri".
If the number of a non-existent dedicated register is given as "Rs", undefined data will be transferred.

■ MOV (Move Word Data in Source Register to Destination Register)

Assembler format: MOV Rs, Ri

Operation: Rs → Ri

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

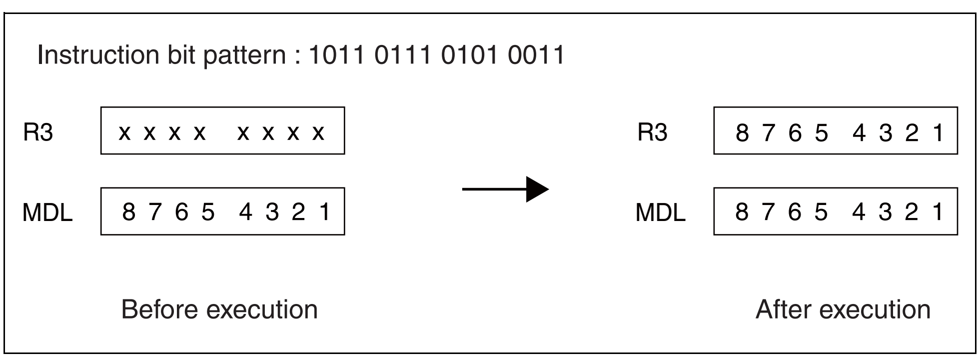
Execution cycles: 1 cycle

Instruction format:

MSB ... LSB

1	0	1	1	0	1	1	1	Rs	Ri

Example: MOV MDL, R3



7.84 MOV (Move Word Data in Program Status Register to Destination Register)

Moves the word data in the program status (PS) to general-purpose register "Ri".

■ MOV (Move Word Data in Program Status Register to Destination Register)

Assembler format: MOV PS, Ri

Operation: PS → Ri

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

Execution cycles: 1 cycle

Instruction format:

MSB												LSB
0	0	0	1	0	1	1	1	0	0	0	1	Ri

Example: MOV PS, R3

Instruction bit pattern : 0001 0111 0001 0011

	Before execution		After execution
R3	x x x x x x x x	R3	F F F 8 F 8 C 0
PS	F F F 8 F 8 C 0	PS	F F F 8 F 8 C 0

7.85 MOV (Move Word Data in Source Register to Destination Register)

Moves the word data in general-purpose register "Ri" to dedicated register "Rs".
If the number of a non-existent register is given as parameter "Rs", the read value "Ri" will be ignored.

■ MOV (Move Word Data in Source Register to Destination Register)

Assembler format: MOV Ri, Rs

Operation: Ri → Rs

Flag change:

N	Z	V	C
–	–	–	–

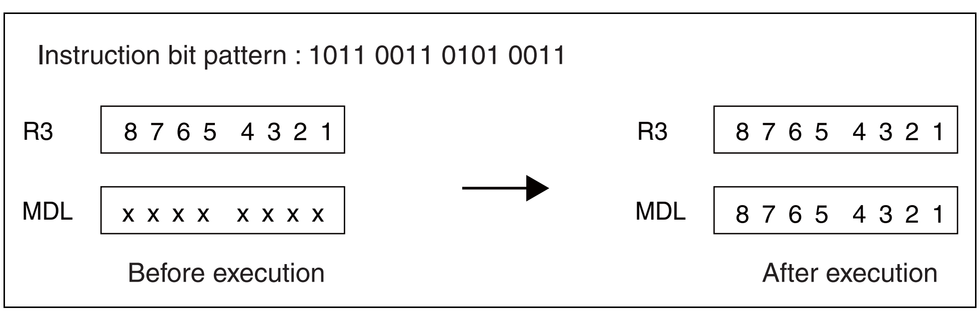
N, Z, V, and C: Unchanged

Execution cycles: 1 cycle

Instruction format:

MSB									LSB
1	0	1	1	0	0	1	1	Rs	Ri

Example: MOV R3, MDL



7.86 MOV (Move Word Data in Source Register to Program Status Register)

Moves the word data in general-purpose register Ri to the program status (PS).
At the time this instruction is executed, if the value of the interrupt level mask register (ILM) is in the range 16 to 31, only new "ILM" settings between 16 and 31 can be entered. If data in the range 0 to 15 is loaded from "Ri", the value 16 will be added to that data before being transferred to the "ILM". If the original "ILM" value is in the range 0 to 15, then any value from 0 to 31 can be transferred to the "ILM".

■ MOV (Move Word Data in Source Register to Program Status Register)

Assembler format: MOV Ri, PS

Operation: Ri → PS

Flag change:

N	Z	V	C
C	C	C	C

N, Z, V, and C: Data is transferred from "Ri".

Execution cycles: c cycle(s)

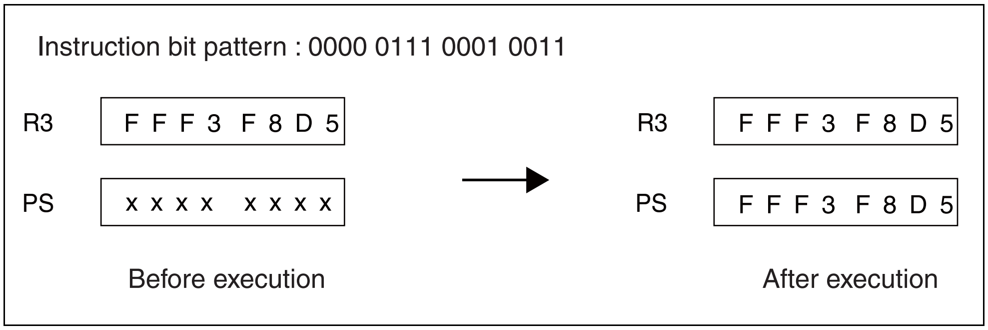
The number of execution cycles is normally "1". However, if the instruction immediately after involves read or write access to memory address "R15", the system stack pointer (SSP) or the user stack pointer (USP), then an interlock is applied and the value becomes 2 cycles.

Instruction format:

MSB ... LSB

0	0	0	0	0	1	1	1	0	0	0	1	Ri

Example: MOV R3, PS



7.87 JMP (Jump)

This is a branching instruction with no delay slot.
Branches to the address indicated by "Ri".

■ JMP (Jump)

Assembler format: JMP @Ri

Operation: Ri → PC

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

Execution cycles: 2 cycles

Instruction format:

MSB ... LSB

1	0	0	1	0	1	1	1	0	0	0	0	Ri

Example: JMP @R1

Instruction bit pattern : 1001 0111 0000 0001

	Before execution		After execution
R1	C 0 0 0 8 0 0 0	→	0 0 0 0 0 0 F F
PC	F F 8 0 0 0 0 0	→	C 0 0 0 8 0 0 0

7.88 CALL (Call Subroutine)

This is a branching instruction with no delay slot.
After storing the address of the next instruction in the return pointer (RP), branch to the address indicated by "lavel12" relative to the value of the program counter (PC). When calculating the address, double the value of "rel11" as a signed extension.

■ CALL (Call Subroutine)

Assembler format: CALL label12

Operation: PC + 2 → RP
PC +2 + exts (rel11 × 2) → PC

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

Execution cycles: 2 cycles

Instruction format:

MSB					LSB
1	1	0	1	0	rel11

Example: CALL label
...
label: ; CALL instruction address + 122_H

Instruction bit pattern : 1101 0000 1001 0000

	Before execution		After execution
PC	FF80 0000	→	FF80 0122
RP	xxxx xxxx		FF80 0004

7.89 CALL (Call Subroutine)

This is a branching instruction with no delay slot.
After storing the address of the next instruction in the return pointer (RP), a branch to the address indicated by "Ri" occurs.

■ CALL (Call Subroutine)

Assembler format: CALL @Ri

Operation: PC + 2 → RP
Ri → PC

Flag change:

N	Z	V	C
–	–	–	–

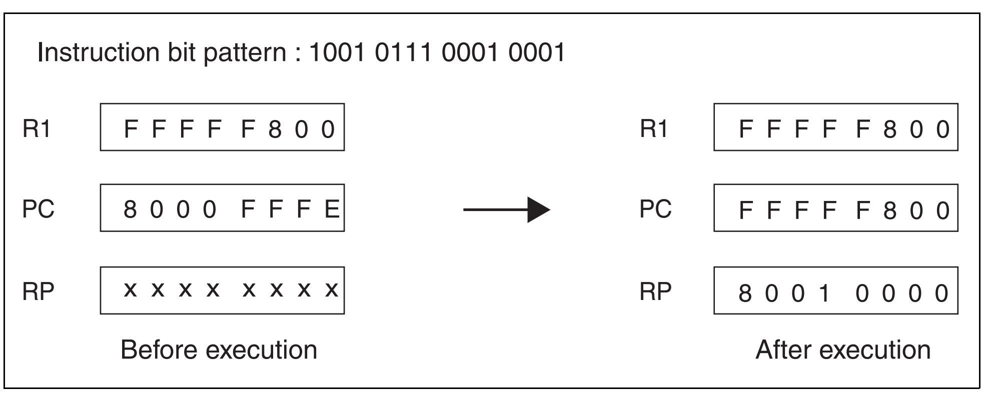
N, Z, V, and C: Unchanged

Execution cycles: 2 cycles

Instruction format:

MSB												LSB
1	0	0	1	0	1	1	1	0	0	0	1	Ri

Example: CALL @R1



7.90 RET (Return from Subroutine)

This is a branching instruction with no delay slot.
Branches to the address indicated by the return pointer (RP).

■ RET (Return from Subroutine)

Assembler format: RET

Operation: RP → PC

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

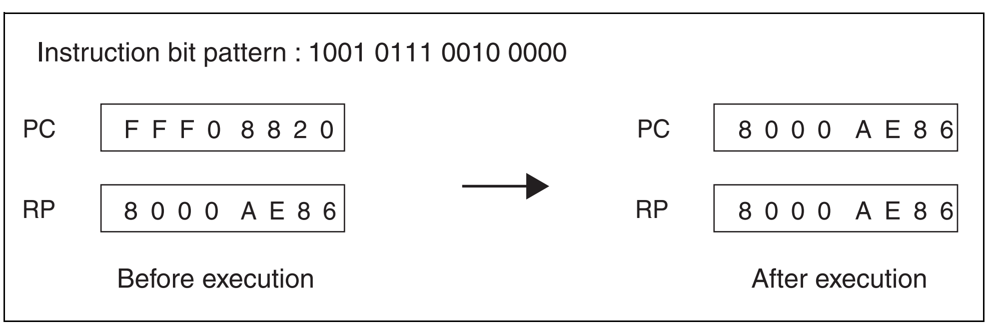
Execution cycles: 2 cycles

Instruction format:

MSB … LSB

1	0	0	1	0	1	1	1	0	0	1	0	0	0	0	0

Example: RET



7.91 INT (Software Interrupt)

Stores the values of the program counter (PC) and program status (PS) to the stack indicated by the system stack pointer (SSP) for interrupt processing. Writes "0" to the "S" flag in the condition code register (CCR), and uses the "SSP" as the stack pointer for the following steps. Writes "0" to the "I" flag (interrupt enable flag) in the "CCR" to disable external interrupts. Reads the vector table for the interrupt vector number "u8" to determine the branch destination address, and branches.
This instruction has no delay slot.
Vector numbers 9 to 13, 64 and 65 are used by emulators for debugging interrupts and therefore the corresponding numbers "INT#9" to "INT#13", "INT#64", "INT#65" should not be used in user programs.

■ INT (Software Interrupt)

Assembler format: INT #u8

Operation:
SSP – 4 → SSP
PS → (SSP)
SSP – 4 → SSP
PC + 2 → (SSP)
"0" → I flag
"0" → S flag
(TBR + $3FC_H$ – u8 × 4) → PC

Flag change:

S	I	N	Z	V	C
0	0	–	–	–	–

N, Z, V, and C: Unchanged

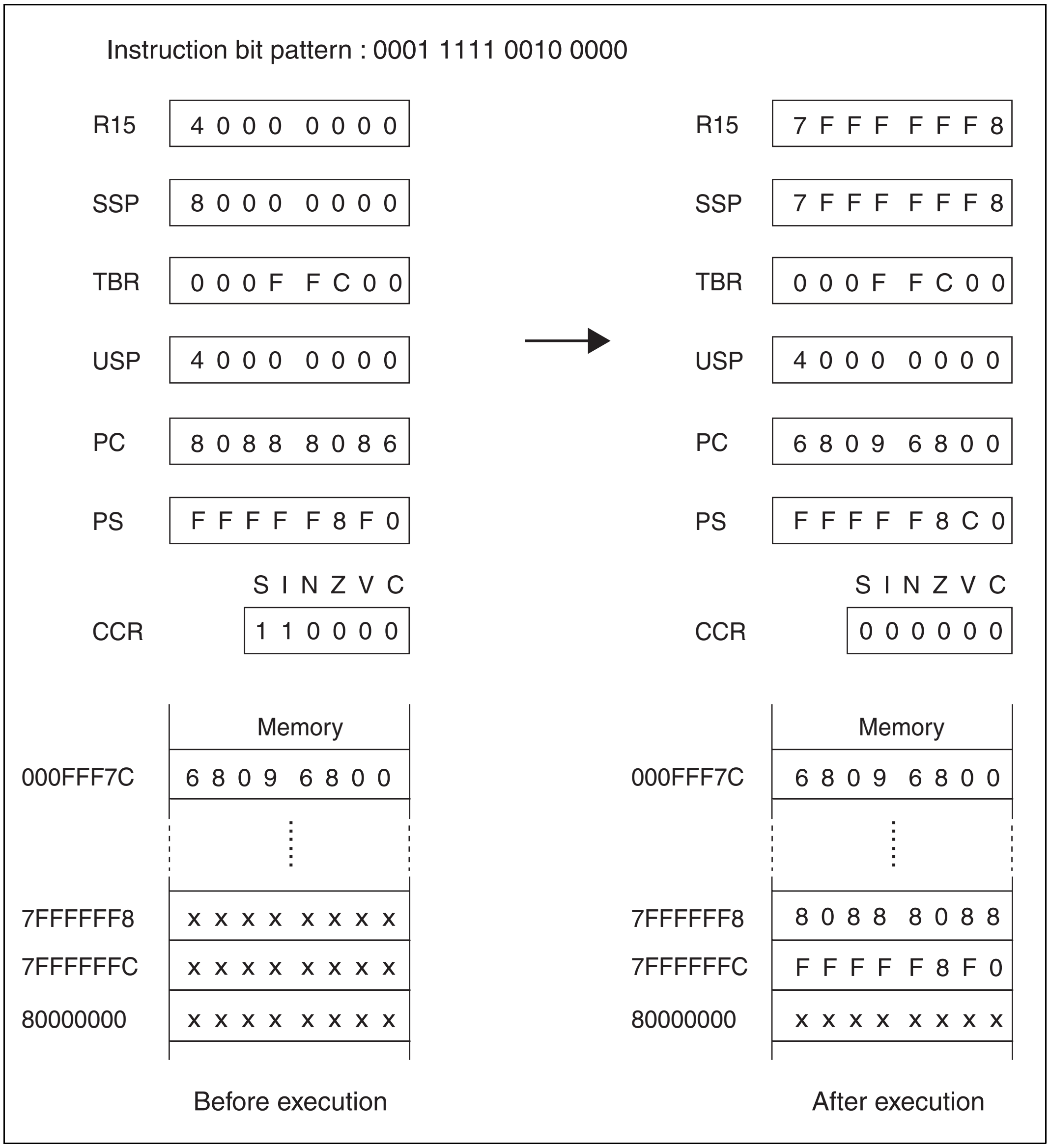
S and I: Cleared to "0".

Execution cycles: 3 + 3a cycles

Instruction format:

MSB								LSB
0	0	0	1	1	1	1	1	u8

Example: INT #20H



7.92 INTE (Software Interrupt for Emulator)

This software interrupt instruction is used for debugging. It stores the values of the program counter (PC) and program status (PS) to the stack indicated by the system stack pointer (SSP) for interrupt processing. It writes "0" to the "S" flag in the condition code register (CCR), and uses the "SSP" as the stack pointer for the following steps.
It determines the branch destination address by reading interrupt vector number "#9" from the vector table, then branches.
There is no change to the "I" flag in the condition code register (CCR).
The interrupt level mask register (ILM) in the program status (PS) is set to level 4.
This instruction is the software interrupt instruction for debugging.
In step execution, no "EIT" events are generated by the "INTE" instruction.
This instruction has no delay slot.

■ INTE (Software Interrupt for Emulator)

Assembler format: INTE

Operation:
SSP – 4 → SSP
PS → (SSP)
SSP – 4 → SSP
PC + 2 → (SSP)
4 → ILM
"0" → S flag
(TBR + $3D8_H$) → PC

Flag change:

S	I	N	Z	V	C
0	–	–	–	–	–

I, N, Z, V, and C: Unchanged
S: Cleared to "0".

Execution cycles: 3 + 3a cycles

Instruction format:

MSB															LSB
1	0	0	1	1	1	1	1	0	0	1	1	0	0	0	0

Example: INTE

Instruction bit pattern : 1001 1111 0011 0000

Register	Before execution	After execution
R15	4000 0000	7FFF FFF8
SSP	8000 0000	7FFF FFF8
USP	4000 0000	4000 0000
TBR	000F FC00	000F FC00
PC	8088 8086	6809 6800
PS	FFF5 F8F0	FFE4 F8D0
ILM	10101	00100
CCR (S I N Z V C)	110000	010000

Memory address	Before execution	After execution
000FFFD8	6809 6800	6809 6800
⋮	⋮	⋮
7FFFFFF8	xxxx xxxx	8088 8088
7FFFFFFC	xxxx xxxx	FFFF F8F0
80000000	xxxx xxxx	xxxx xxxx

7.93 RETI (Return from Interrupt)

Loads data from the stack indicated by "R15" to the program counter (PC) and program status (PS), and retakes control from the interrupt handler.
This instruction requires the S flag in the register (CCR) to be executed in a state of "0". Do not manipulate the S flag in the normal interrupt handler; use it in a state of 0 as it is.
This instruction has no delay slot.
At the time this instruction is executed, if the value of the interrupt level mask register (ILM) is in the range 16 to 31, only new "ILM" settings between 16 and 31 can be entered. If data in the range 0 to 15 is loaded in memory, the value 16 will be added to that data before being transferred to the "ILM". If the original "ILM" value is in the range 0 to 15, then any value between 0 and 31 can be transferred to the "ILM".

■ RETI (Return from Interrupt)

Assembler format: RETI

Operation:
(R15) → PC
R15 + 4 → R15
(R15) → PS
R15 + 4 →R15

Flag change:

S	I	N	Z	V	C
C	C	C	C	C	C

S, I, N, Z, V, and C: Change according to values retrieved from the stack.

Execution cycles: 2 + 2a cycles

Instruction format:

MSB															LSB
1	0	0	1	0	1	1	1	0	0	1	1	0	0	0	0

Example: RETI

Instruction bit pattern : 1001 0111 0011 0000

	Before execution	After execution
R15	7 F F F F F F 8	4 0 0 0 0 0 0 0
SSP	7 F F F F F F 8	8 0 0 0 0 0 0 0
USP	4 0 0 0 0 0 0 0	4 0 0 0 0 0 0 0
PC	F F 0 0 9 0 B C	8 0 8 8 8 0 8 8
PS	F F F 0 F 8 D 4	F F F 3 F 8 F 1
ILM	1 0 0 0 0	1 0 0 1 1
CCR (S I N Z V C)	0 1 0 1 0 0	1 1 0 0 0 1

Memory	Before execution	After execution
7FFFFFF8	8 0 8 8 8 0 8 8	8 0 8 8 8 0 8 8
7FFFFFFC	F F F 3 F 8 F 1	F F F 3 F 8 F 1
80000000	x x x x x x x x	x x x x x x x x

7.94 Bcc (Branch Relative if Condition Satisfied)

This branching instruction has no delay slot.
If the conditions established for each particular instruction are satisfied, branch to the address indicated by "label9" relative to the value of the program counter (PC). When calculating the address, double the value of "rel8" as a signed extension.
If conditions are not satisfied, no branching can occur.
Conditions for each instruction are listed in Table 7.94-1.

■ Bcc (Branch Relative if Condition Satisfied)

Assembler format:

```
BRA  label9     BV   label9
BNO  label9     BNV  label9
BEQ  label9     BLT  label9
BNE  label9     BGE  label9
BC   label9     BLE  label9
BNC  label9     BGT  label9
BN   label9     BLS  label9
BP   label9     BHI  label9
```

Operation:

```
if (conditions satisfied) {
PC + 2 + exts (rel8 × 2) → PC
}
```

Table 7.94-1 Branching Conditions

Mnemonic	cc	Conditions	Mnemonic	cc	Conditions
BRA	0000	Always satisfied	BV	1000	V = 1
BNO	0001	Always unsatisfied	BNV	1001	V = 0
BEQ	0010	Z = 1	BLT	1010	V xor N = 1
BNE	0011	Z = 0	BGE	1011	V xor N = 0
BC	0100	C = 1	BLE	1100	(V xor N) or Z = 1
BNC	0101	C = 0	BGT	1101	(V xor N) or Z = 0
BN	0110	N = 1	BLS	1110	C or Z = 1
BP	0111	N = 0	BHI	1111	C or Z = 0

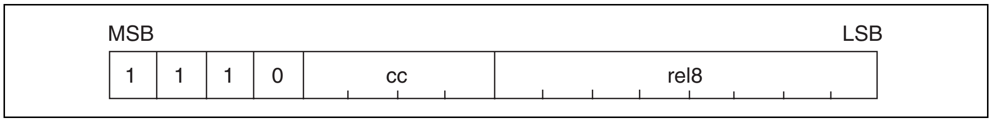
Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

Execution cycles: Branch: 2 cycles
Not branch: 1 cycle

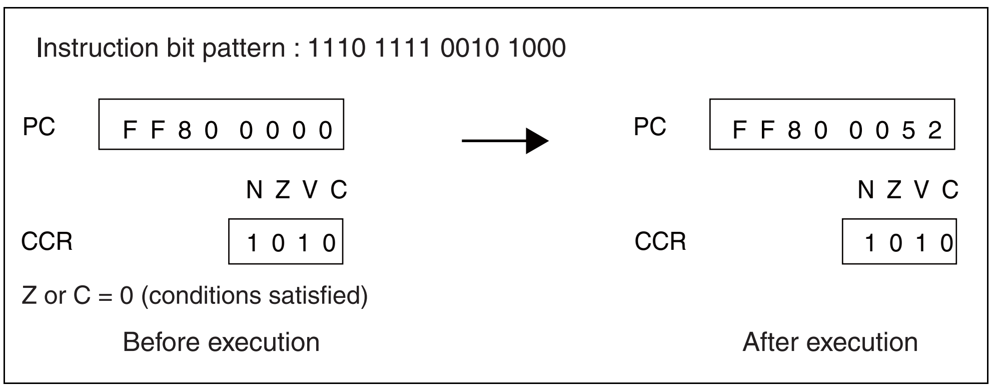
Instruction format:



Example: BHI label

...

label: ; BHI instruction address + 50_H



7.95 JMP:D (Jump)

This branching instruction has a delay slot.
Branches to the address indicated by "Ri".

■ JMP:D (Jump)

Assembler format: JMP : D @Ri

Operation: Ri → PC

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

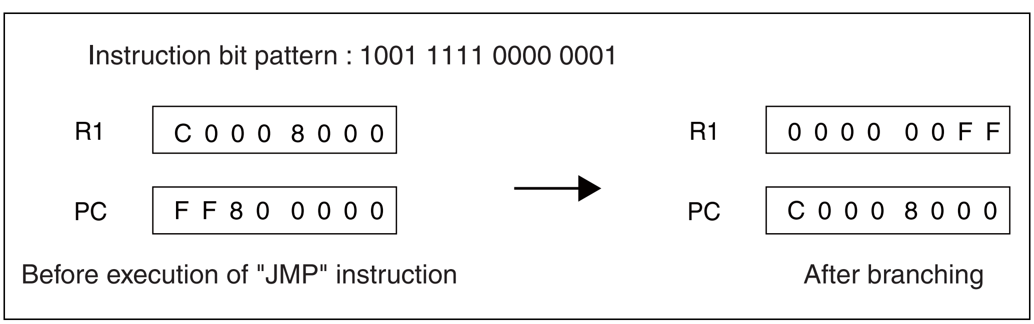
Execution cycles: 1 cycle

Instruction format:

MSB												LSB
1	0	0	1	1	1	1	1	0	0	0	0	Ri

Example:

```
JMP : D @R1
LDI : 8  #0FFH, R1                ; Instruction placed in delay slot
  ⋮
```



The instruction placed in the delay slot will be executed before execution of the branch destination instruction.

The value "R1" above will vary according to the specifications of the "LDI:8" instruction placed in the delay slot.

7.96 CALL:D (Call Subroutine)

This is a branching instruction with a delay slot.
After saving the address of the next instruction after the delay slot to the "RP", branch to the address indicated by "label12" relative to the value of the program counter (PC).
When calculating the address, double the value of "rel11" as a signed extension.

■ CALL:D (Call Subroutine)

Assembler format: CALL : D label12

Operation: PC + 4 → RP
PC + 2 + exts (rel11 × 2) → PC

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

Execution cycles: 1 cycle

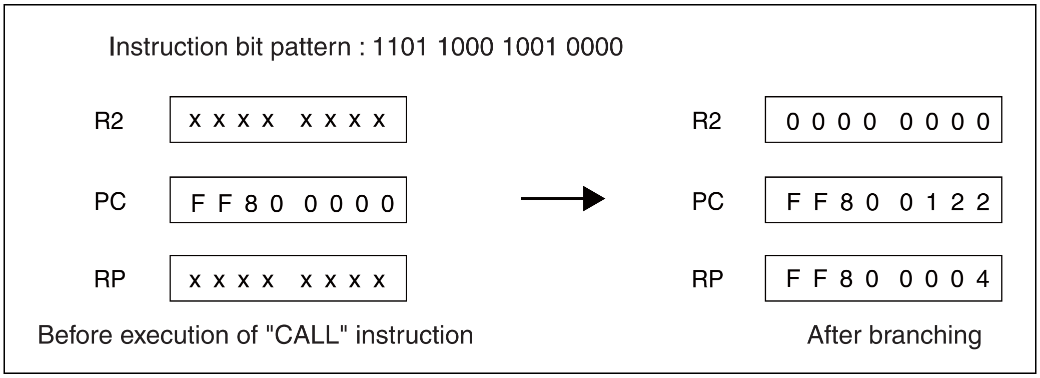
Instruction format:

MSB ... LSB

1	1	0	1	1	rel11

Example:

```
CALL:D  label
LDI : 8  #0, R2                    ; Instruction placed in delay slot
...
label:  ; CALL: D instruction address + 122H
```



The instruction placed in the delay slot will be executed before execution of the branch destination instruction.

The value "R2" above will vary according to the specifications of the "LDI:8" instruction placed in the delay slot.

7.97 CALL:D (Call Subroutine)

This is a branching instruction with a delay slot.
After saving the address of the next instruction after the delay slot to the "RP", it branches to the address indicated by "Ri".

■ CALL:D (Call Subroutine)

Assembler format: CALL : D @Ri

Operation: PC + 4 → RP
Ri → PC

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

Execution cycles: 1 cycle

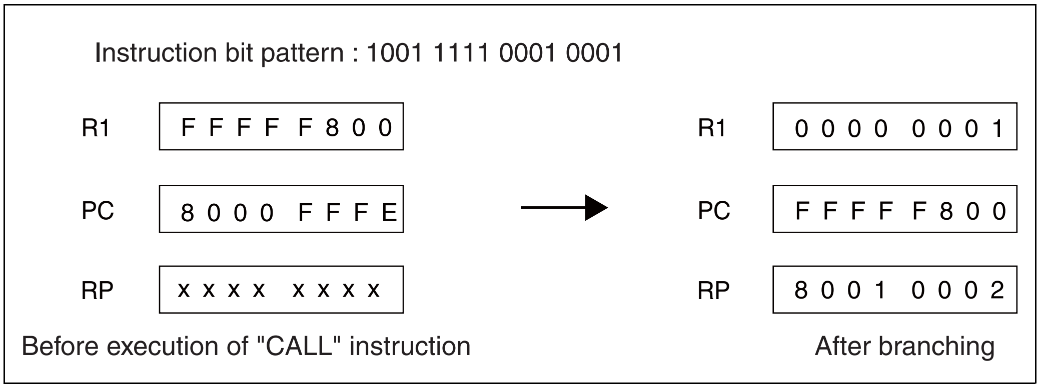
Instruction format:

MSB ... LSB

1	0	0	1	1	1	1	1	0	0	0	1	Ri

Example:

```
CALL : D @R1
LDI : 8  #1, R1                    ; Instruction placed in delay slot
  :
```



The instruction placed in the delay slot will be executed before execution of the branch destination instruction.

The value "R1" above will vary according to the specifications of the "LDI:8" instruction placed in the delay slot.

7.98 RET:D (Return from Subroutine)

This is a branching instruction with a delay slot.
Branches to the address indicated by the "RP".

■ RET:D (Return from Subroutine)

Assembler format: RET : D

Operation: RP → PC

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

Execution cycles: 1 cycle

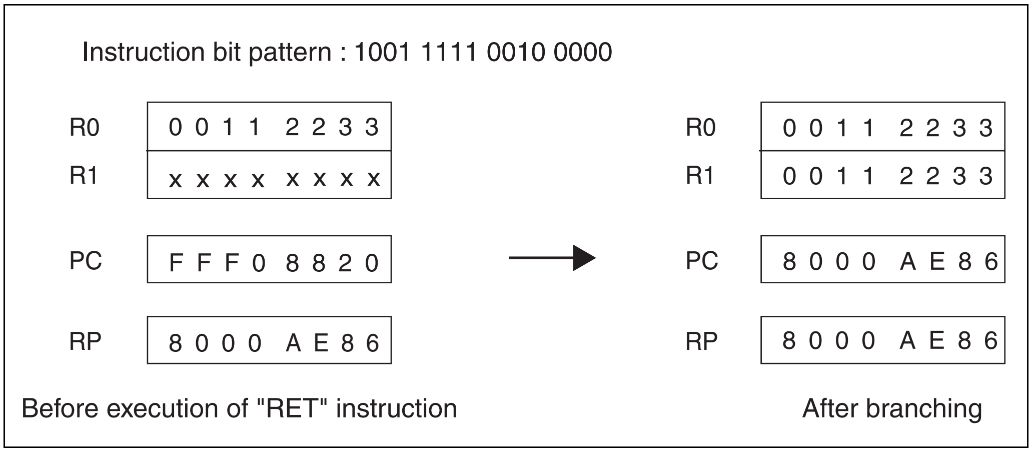
Instruction format:

MSB ... LSB

1	0	0	1	1	1	1	1	0	0	1	0	0	0	0	0

Example:

```
RET : D
MOV  R0, R1     ; Instruction placed in delay slot
  ⋮
```



The instruction placed in the delay slot will be executed before execution of the branch destination instruction.

The value "R1" above will vary according to the specifications of the "MOV" instruction placed in the delay slot.

7.99 Bcc:D (Branch Relative if Condition Satisfied)

This is a branching instruction with a delay slot.
If the conditions established for each particular instruction are satisfied, branch to the address indicated by "label9" relative to the value of the program counter (PC). When calculating the address, double the value of "rel8" as a signed extension.
If conditions are not satisfied, no branching can occur.
Conditions for each instruction are listed in Table 7.99-1.

■ Bcc:D (Branch Relative if Condition Satisfied)

Assembler format:

BRA : D	label9	BV : D	label9
BNO : D	label9	BNV : D	label9
BEQ : D	label9	BLT : D	label9
BNE : D	label9	BGE : D	label9
BC : D	label9	BLE : D	label9
BNC : D	label9	BGT : D	label9
BN : D	label9	BLS : D	label9
BP : D	label9	BHI : D	label9

Operation:

if (conditions satisfied) {
  PC + 2 + exts (rel8 × 2) → PC
}

Table 7.99-1 Branching Conditions

Mnemonic	cc	Conditions	Mnemonic	cc	Conditions
BRA : D	0000_B	Always satisfied	BV : D	1000_B	V = 1
BNO : D	0001_B	Always unsatisfied	BNV : D	1001_B	V = 0
BEQ : D	0010_B	Z = 1	BLT : D	1010_B	V xor N = 1
BNE : D	0011_B	Z = 0	BGE : D	1011_B	V xor N = 0
BC : D	0100_B	C = 1	BLE : D	1100_B	(V xor N) or Z = 1
BNC : D	0101_B	C = 0	BGT : D	1101_B	(V xor N) or Z = 0
BN : D	0110_B	N = 1	BLS : D	1110_B	C or Z = 1
BP : D	0111_B	N = 0	BHI : D	1111_B	C or Z = 0

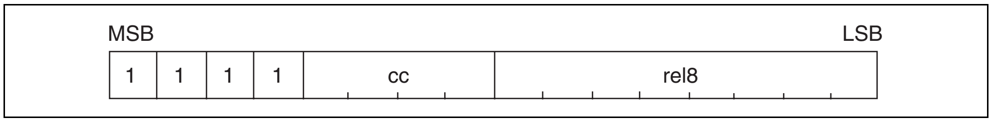
Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

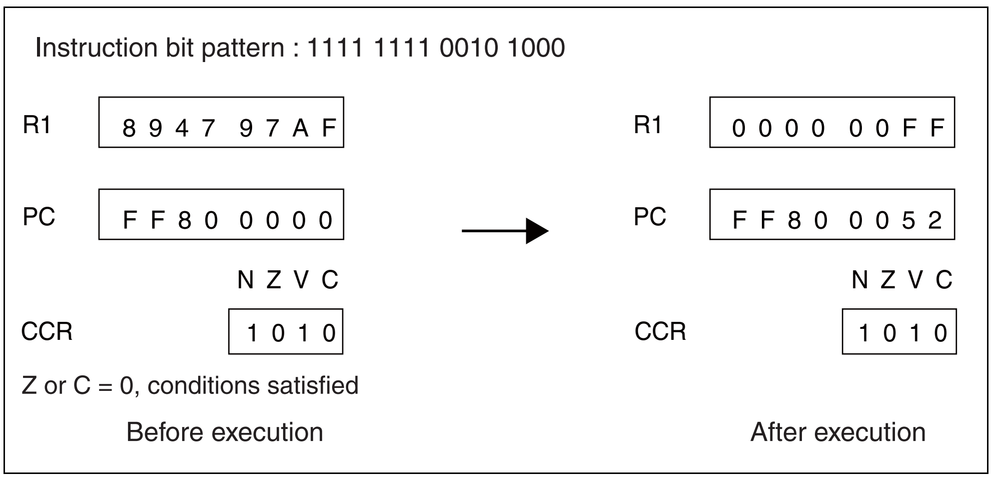
Execution cycles: 1 cycle

Instruction format:



Example:

BHI:D label
LDI :8 #255, R1 ; Instruction placed in delay slot
⋮
label: ; BHI: D instruction address + 50_H



The instruction placed in the delay slot will be executed before execution of the branch destination instruction.

The value "R1" above will vary according to the specifications of the "LDI:8" instruction placed in the delay slot.

7.100 DMOV (Move Word Data from Direct Address to Register)

Transfers, to "R13", the word data at the direct address corresponding to 4 times the value of "dir8".

■ DMOV (Move Word Data from Direct Address to Register)

Assembler format: DMOV @dir10, R13

Operation: (dir8 × 4) → R13

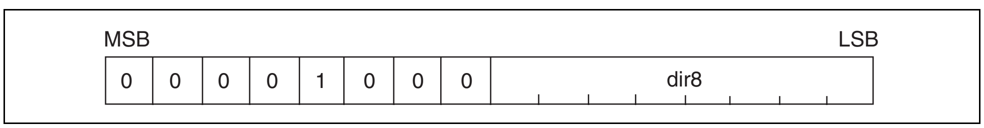
Flag change:

N	Z	V	C
–	–	–	–

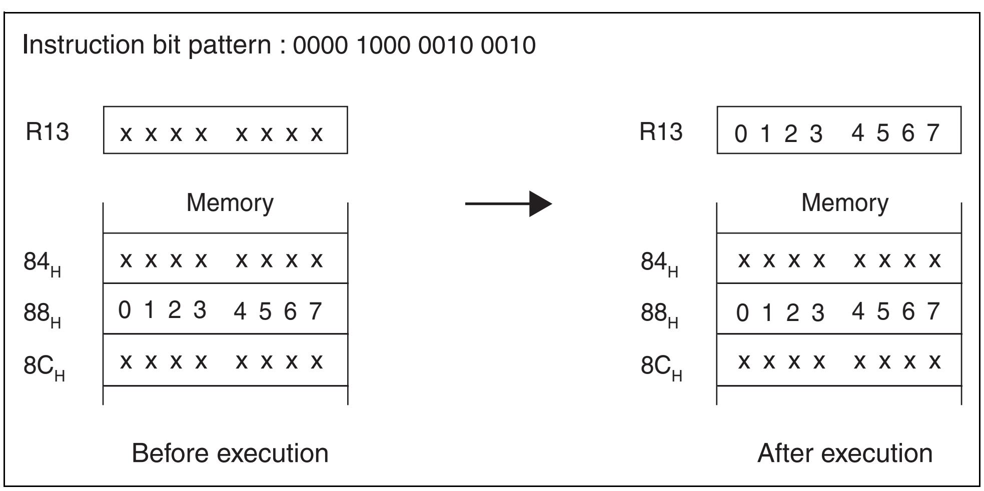
N, Z, V, and C: Unchanged

Execution cycles: b cycle(s)

Instruction format:



Example: DMOV @88H, R13



7.101 DMOV (Move Word Data from Register to Direct Address)

Transfers the word data in "R13" to the direct address corresponding to 4 times the value of "dir8".

■ DMOV (Move Word Data from Register to Direct Address)

Assembler format: DMOV R13, @dir10

Operation: R13 → (dir8 × 4)

Flag change:

N	Z	V	C
–	–	–	–

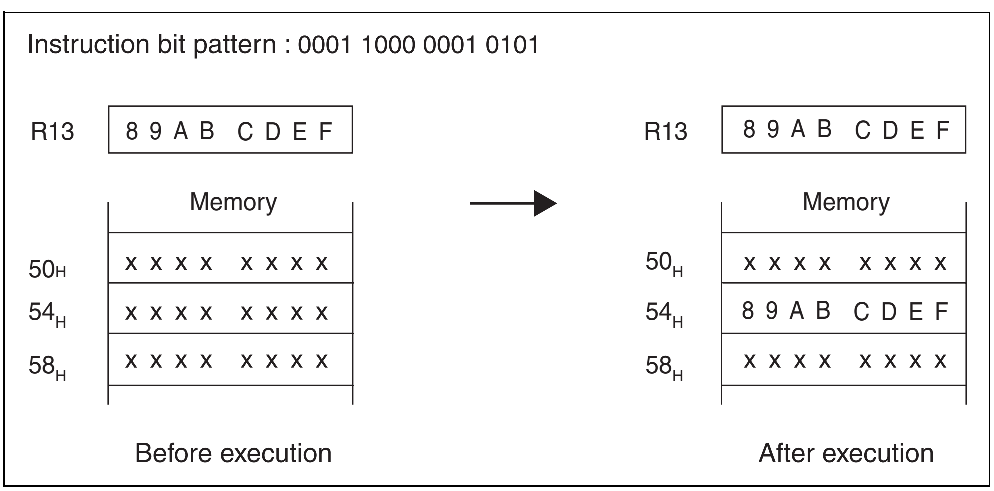
N, Z, V, and C: Unchanged

Execution cycles: a cycle(s)

Instruction format:

MSB								LSB
0	0	0	1	1	0	0	0	dir8

Example: DMOV R13, @54H



7.102 DMOV (Move Word Data from Direct Address to Post Increment Register Indirect Address)

Transfers the word data at the direct address corresponding to 4 times the value of "dir8" to the address indicated in "R13". After the data transfer, it increments the value of "R13" by 4.

■ DMOV (Move Word Data from Direct Address to Post Increment Register Indirect Address)

Assembler format: DMOV @dir10, @R13+

Operation: (dir8 × 4) → (R13)
R13 + 4 → R13

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

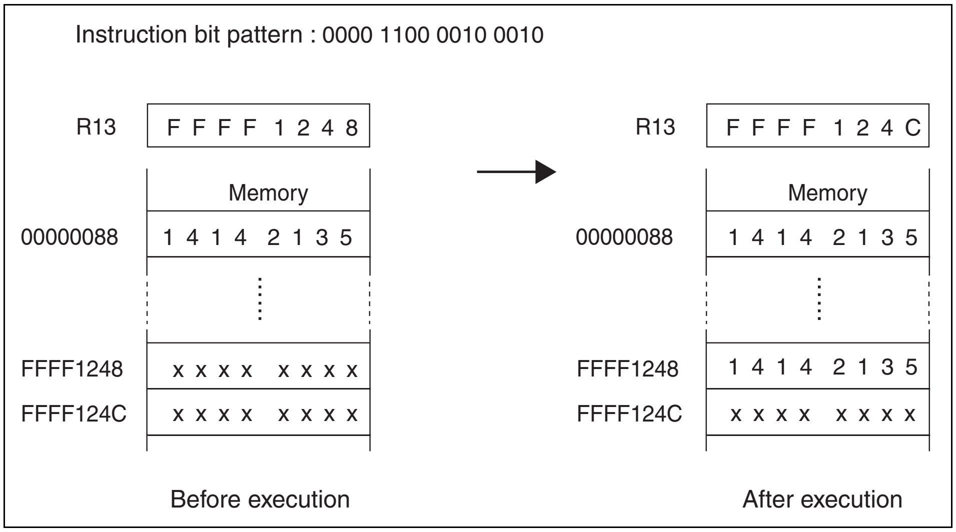
Execution cycles: 2a cycles

Instruction format:

MSB (left) … LSB (right)

0	0	0	0	1	1	0	0	dir8

Example: DMOV @88H, @R13+



7.103 DMOV (Move Word Data from Post Increment Register Indirect Address to Direct Address)

Transfers the word data at the address indicated in "R13" to the direct address corresponding to 4 times the value "dir8". After the data transfer, it increments the value of "R13" by 4.

■ DMOV (Move Word Data from Post Increment Register Indirect Address to Direct Address)

Assembler format: DMOV @R13+, @dir10

Operation: (R13) → (dir8 × 4)
R13 + 4 → R13

Flag change:

N	Z	V	C
–	–	–	–

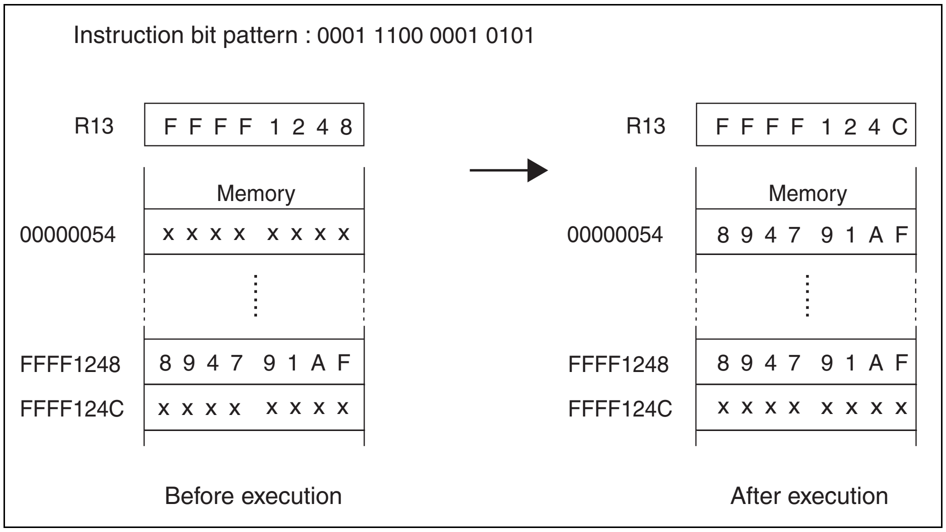
N, Z, V, and C: Unchanged

Execution cycles: 2a cycles

Instruction format:

MSB								LSB
0	0	0	1	1	1	0	0	dir8

Example: DMOV @R13+, @54H



7.104 DMOV (Move Word Data from Direct Address to Pre-decrement Register Indirect Address)

Decrements the value of "R15" by 4, then transfers word data at the direct address corresponding to 4 times the value of "dir8" to the address indicated in "R15".

■ DMOV (Move Word Data from Direct Address to Pre-decrement Register Indirect Address)

Assembler format: DMOV @dir10, @ – R15

Operation: R15 – 4 → R15
(dir8 × 4) → (R15)

Flag change:

N	Z	V	C
–	–	–	–

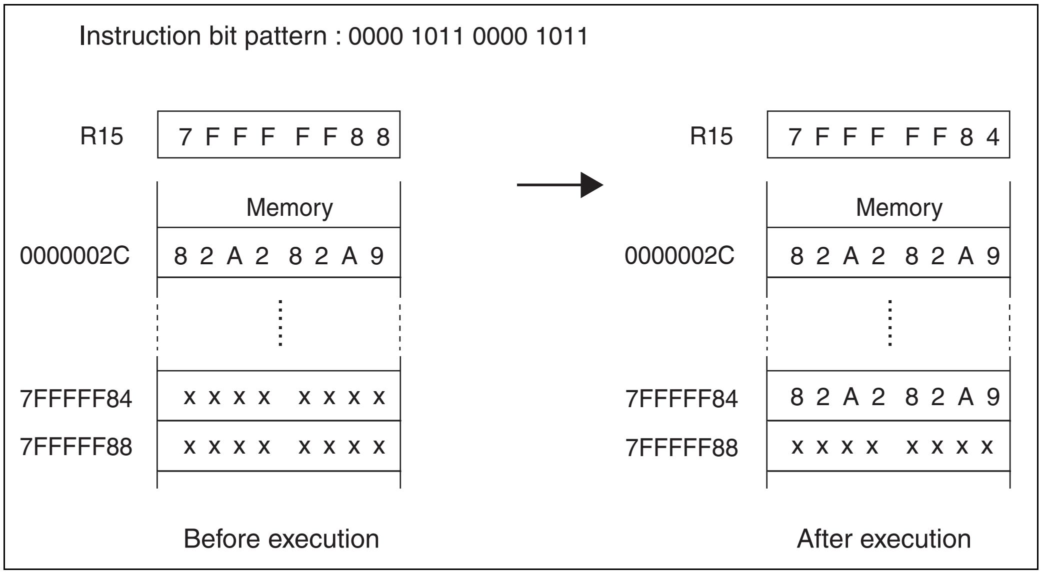
N, Z, V, and C: Unchanged

Execution cycles: 2a cycles

Instruction format:

MSB								LSB
0	0	0	0	1	0	1	1	dir8

Example: DMOV @2CH, @ – R15



7.105 DMOV (Move Word Data from Post Increment Register Indirect Address to Direct Address)

Transfers the word data at the address indicated in "R15" to the direct address corresponding to 4 times the value "dir8". After the data transfer, it increments the value of "R15" by 4.

■ DMOV (Move Word Data from Post Increment Register Indirect Address to Direct Address)

Assembler format: DMOV @R15+, @dir10

Operation: (R15) → (dir8 × 4)
R15 + 4 → R15

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

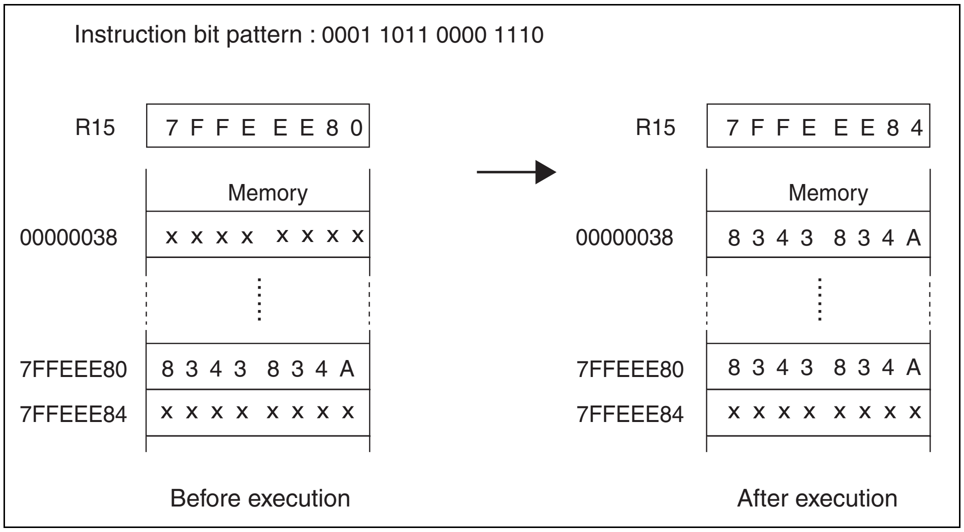
Execution cycles: 2a cycles

Instruction format:

MSB ... LSB

0	0	0	1	1	0	1	1	dir8

Example: DMOV @R15+, @38H



7.106 DMOVH (Move Half-word Data from Direct Address to Register)

Transfers the half-word data at the direct address corresponding to 2 times the value "dir8" to "R13". Uses zeros to extend the higher 16 bits of data.

■ DMOVH (Move Half-word Data from Direct Address to Register)

Assembler format: DMOVH @dir9, R13

Operation: (dir8 × 2) → R13

Flag change:

N	Z	V	C
–	–	–	–

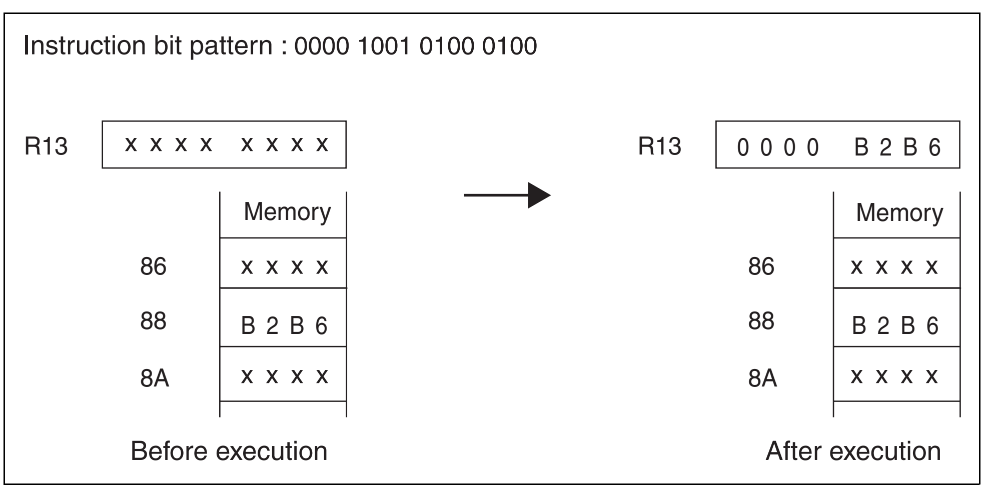
N, Z, V, and C: Unchanged

Execution cycles: b cycle(s)

Instruction format:

MSB								LSB
0	0	0	0	1	0	0	1	dir8

Example: DMOVH @88H, R13



7.107 DMOVH (Move Half-word Data from Register to Direct Address)

Transfers the half-word data from "R13" to the direct address corresponding to 2 times the value "dir8".

■ DMOVH (Move Half-word Data from Register to Direct Address)

Assembler format: DMOVH R13, @dir9

Operation: R13 → (dir8 × 2)

Flag change:

N	Z	V	C
–	–	–	–

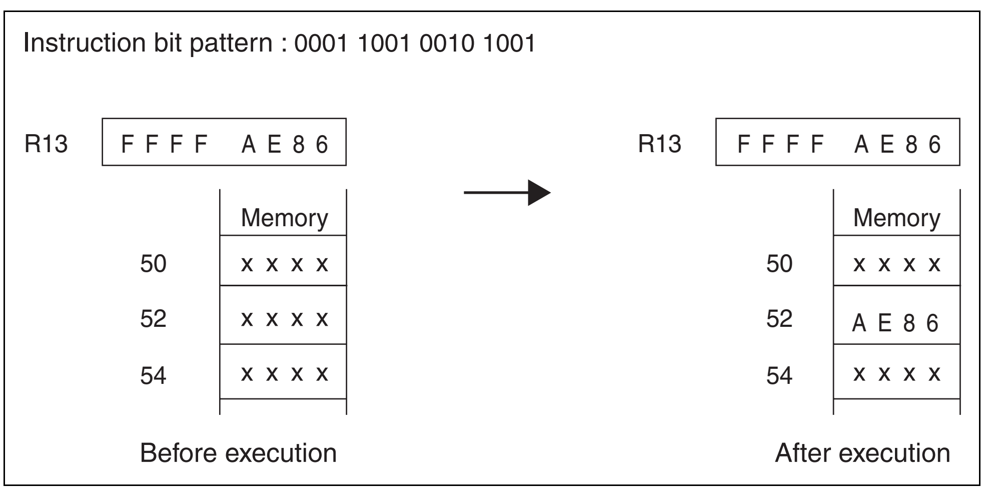
N, Z, V, and C: Unchanged

Execution cycles: a cycle(s)

Instruction format:

MSB								LSB
0	0	0	1	1	0	0	1	dir8

Example: DMOVH R13, @52H



7.108 DMOVH (Move Half-word Data from Direct Address to Post Increment Register Indirect Address)

Transfers the half-word data at the direct address corresponding to 2 times the value "dir8" to the address indicated by "R13". After the data transfer, it increments the value of "R13" by 2.

■ DMOVH (Move Half-word Data from Direct Address to Post Increment Register Indirect Address)

Assembler format: DMOVH @dir9, @R13+

Operation: (dir8 × 2) → (R13)
R13 + 2 → R13

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

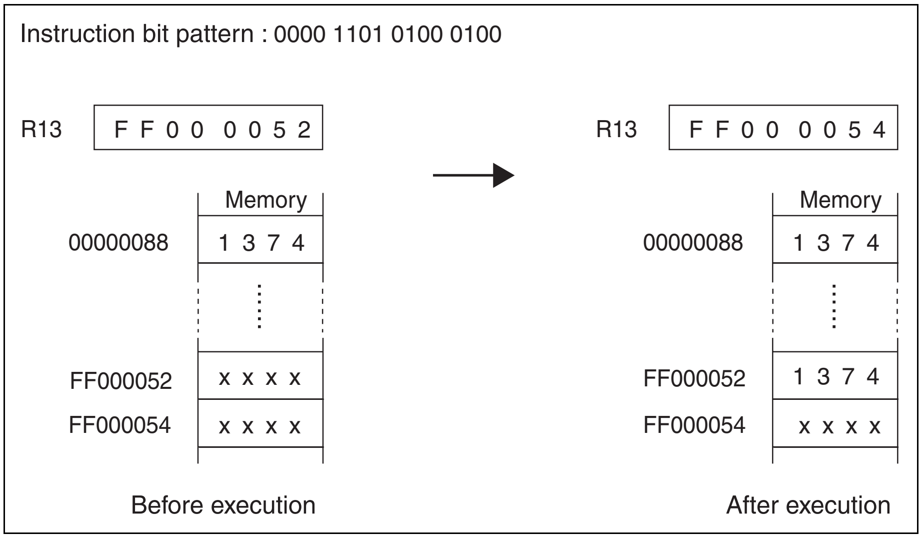
Execution cycles: 2a cycles

Instruction format:

MSB ... LSB

0	0	0	0	1	1	0	1	dir8

Example: DMOVH @88H, @R13+



7.109 DMOVH (Move Half-word Data from Post Increment Register Indirect Address to Direct Address)

Transfers the half-word data at the address indicated by "R13" to the direct address corresponding to 2 times the value "dir8". After the data transfer, it increments the value of "R13" by 2.

■ DMOVH (Move Half-word Data from Post Increment Register Indirect Address to Direct Address)

Assembler format: DMOVH @R13+, @dir9

Operation: (R13) → (dir8 × 2)
R13 + 2 → R13

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

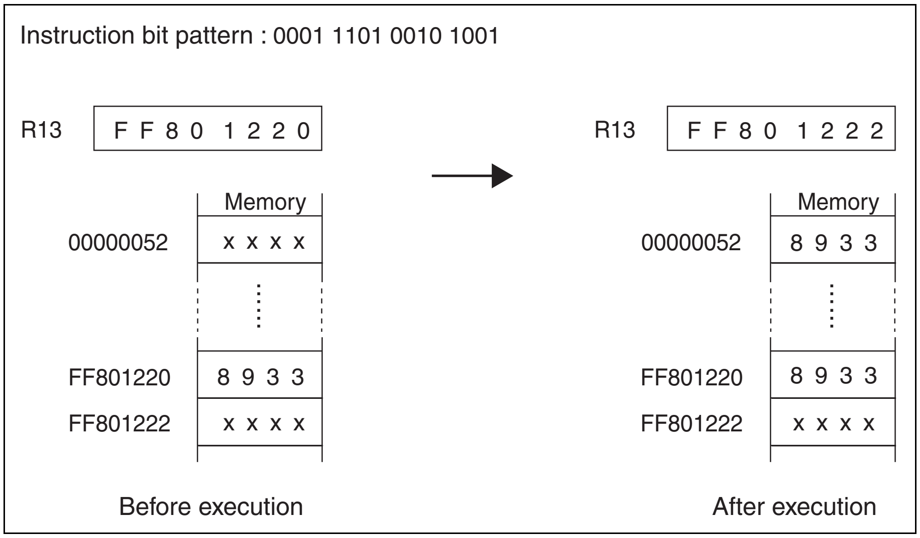
Execution cycles: 2a cycles

Instruction format:

MSB ... LSB

0	0	0	1	1	1	0	1	dir8

Example: DMOVH @R13+, @52H



7.110 DMOVB (Move Byte Data from Direct Address to Register)

Transfers the byte data at the address indicated by the value "dir8" to "R13". Uses zeros to extend the higher 24 bits of data.

■ DMOVB (Move Byte Data from Direct Address to Register)

Assembler format: DMOVB @dir8, R13

Operation: (dir8) → R13

Flag change:

N	Z	V	C
–	–	–	–

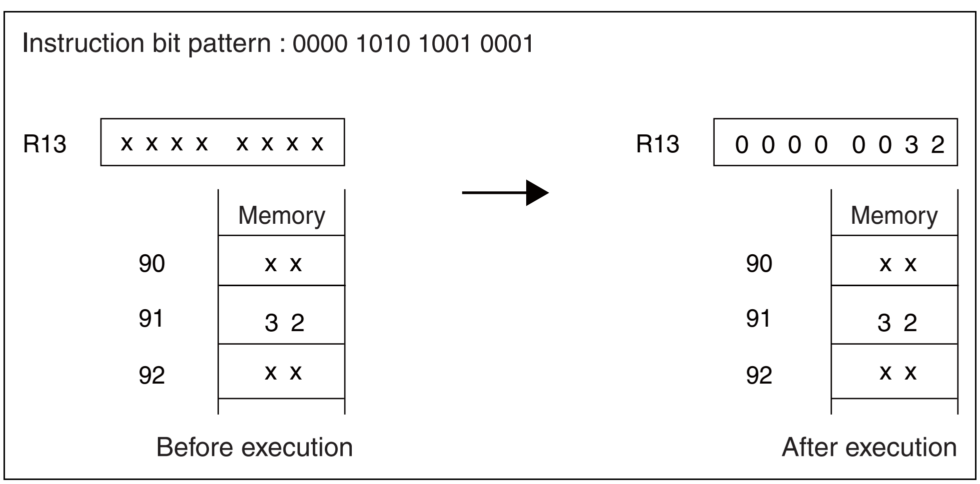
N, Z, V, and C: Unchanged

Execution cycles: b cycle(s)

Instruction format:

MSB								LSB
0	0	0	0	1	0	1	0	dir8

Example: DMOVB @91H, R13



7.111 DMOVB (Move Byte Data from Register to Direct Address)

Transfers the byte data from "R13" to the direct address indicated by the value "dir8".

■ DMOVB (Move Byte Data from Register to Direct Address)

Assembler format: DMOVB R13, @dir8

Operation: R13 → (dir8)

Flag change:

N	Z	V	C
–	–	–	–

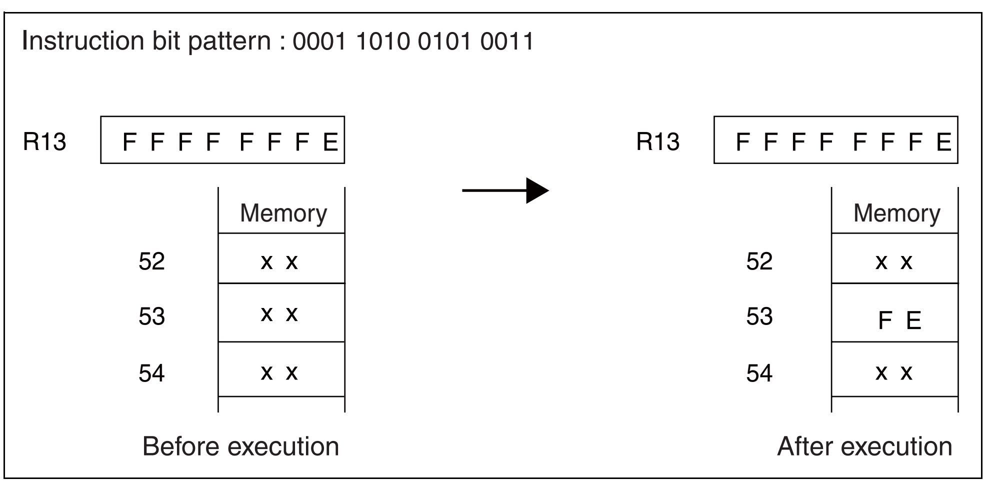
N, Z, V, and C: Unchanged

Execution cycles: a cycle(s)

Instruction format:

MSB								LSB
0	0	0	1	1	0	1	0	dir8

Example: DMOVB R13, @53H



7.112 DMOVB (Move Byte Data from Direct Address to Post Increment Register Indirect Address)

Moves the byte data at the direct address indicated by the value "dir8" to the address indicated by "R13". After the data transfer, it increments the value of "R13" by 1.

■ DMOVB (Move Byte Data from Direct Address to Post Increment Register Indirect Address)

Assembler format: DMOVB @dir8, @R13+

Operation: (dir8) → (R13)
R13 + 1 → R13

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

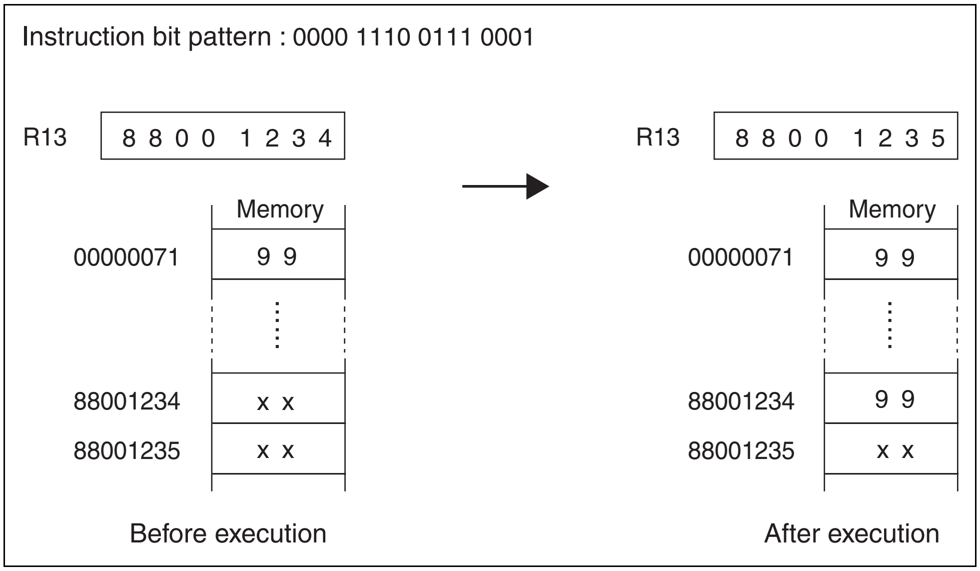
Execution cycles: 2a cycles

Instruction format:

MSB ... LSB

0	0	0	0	1	1	1	0	dir8

Example: DMOVB @71H, @R13+



7.113 DMOVB (Move Byte Data from Post Increment Register Indirect Address to Direct Address)

Transfers the byte data at the address indicated by "R13" to the direct address indicated by the value "dir8". After the data transfer, it increments the value of "R13" by 1.

■ DMOVB (Move Byte Data from Post Increment Register Indirect Address to Direct Address)

Assembler format: DMOVB @R13+, @dir8

Operation: (R13) → (dir8)
R13 + 1 → R13

Flag change:

N	Z	V	C
–	–	–	–

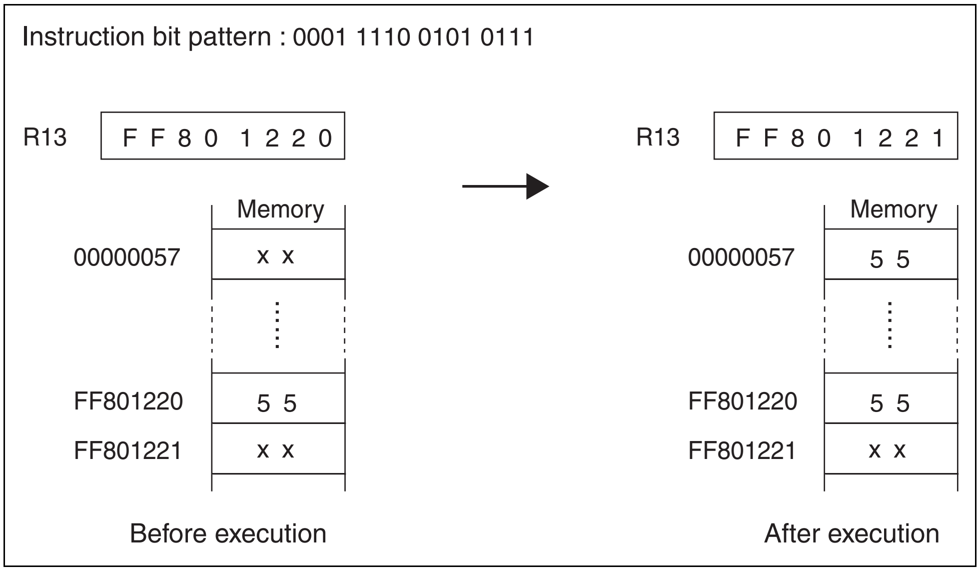
N, Z, V, and C: Unchanged

Execution cycles: 2a cycles

Instruction format:

MSB								LSB
0	0	0	1	1	1	1	0	dir8

Example: DMOVB @R13+, @57H



7.114 LDRES (Load Word Data in Memory to Resource)

Transfers the word data at the address indicated by "Ri" to the resource on channel "u4". Increments the value of "Ri" by 4.

■ LDRES (Load Word Data in Memory to Resource)

Assembler format: LDRES @Ri+, #u4

Operation: (Ri) → Resource on channel u4
Ri + 4 → Ri

Flag change:

N	Z	V	C
–	–	–	–

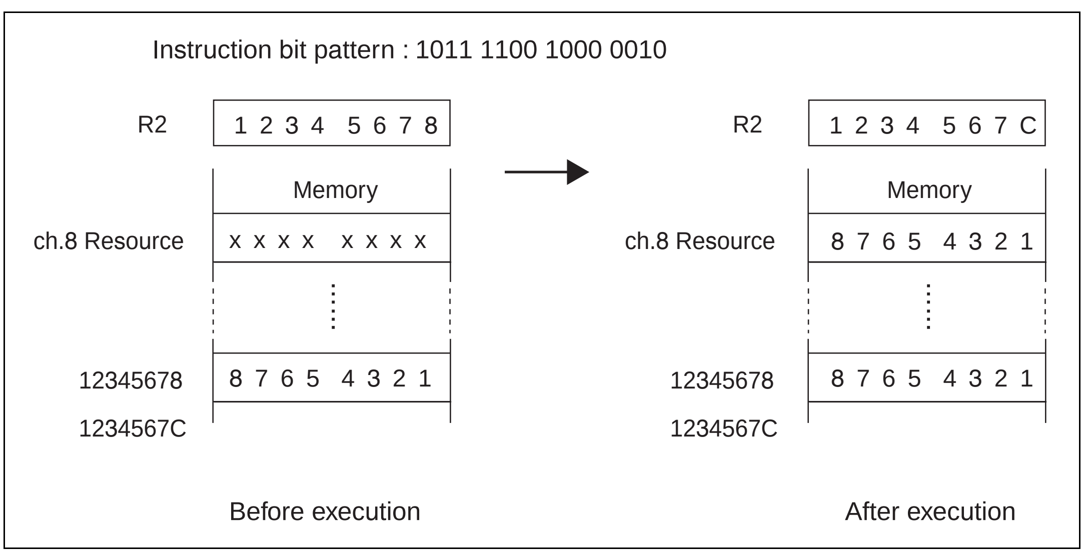
N, Z, V, and C: Unchanged

Execution cycles: a cycle(s)

Instruction format:

MSB									LSB
1	0	1	1	1	1	0	0	u4	Ri

Example: LDRES @R2+, #8



7.115 STRES (Store Word Data in Resource to Memory)

Transfers the word data at the resource on channel "u4" to the address indicated by "Ri". Increments the value of "Ri" by 4.

■ STRES (Store Word Data in Resource to Memory)

Assembler format: STRES #u4, @Ri+

Operation: Resource on channel u4 → (Ri)
Ri + 4 → Ri

Flag change:

N	Z	V	C
–	–	–	–

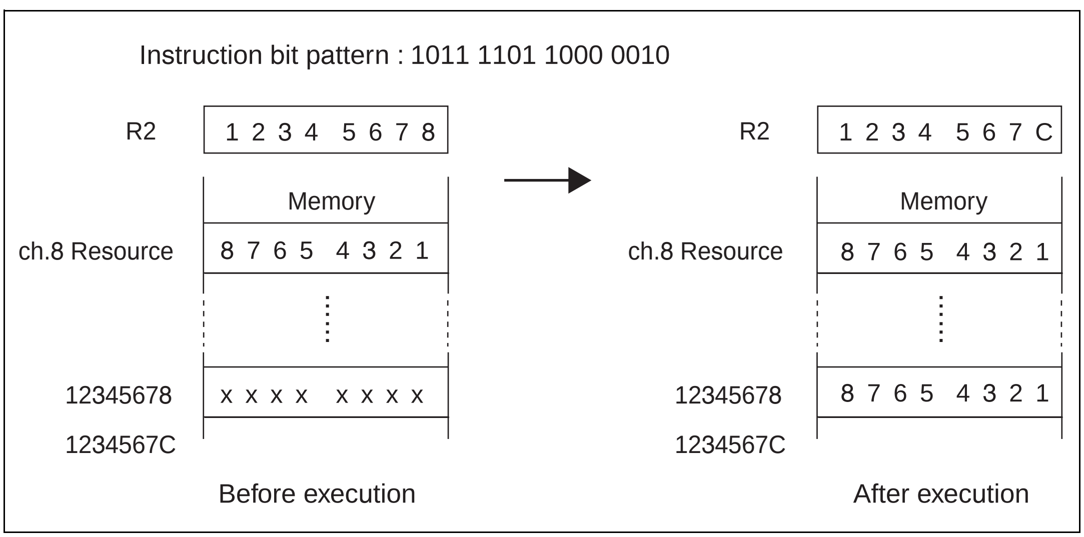
N, Z, V, and C: Unchanged

Execution cycles: a cycle(s)

Instruction format:

MSB									LSB
1	0	1	1	1	1	0	1	u4	Ri

Example: STRES #8, @R2+



7.116 COPOP (Coprocessor Operation)

Transfers the 16-bit data consisting of parameters "CC", "CRj", "CRi" to the coprocessor indicated by channel number "u4".
Basically, this operation is a calculation between registers within the coprocessor. The calculation process indicated by the value "CC" is carried out between coprocessor registers "CRj" and "CRi". Note that the actual interpretation of the fields "CC", "CRj", and "CRi" is done by the coprocessor so that the detailed operation is determined by the specifications of the coprocessor.
If the coprocessor designated by the value "u4" is not mounted, a "coprocessor not found" trap is generated.
If the coprocessor designated by the value "u4" has generated an error in a previous operation, a "coprocessor error" trap is generated.

■ COPOP (Coprocessor Operation)

Assembler format: COPOP #u4, #CC, CRj, CRi

Operation: CC, CRj, CRi → Coprocessor on channel u4

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

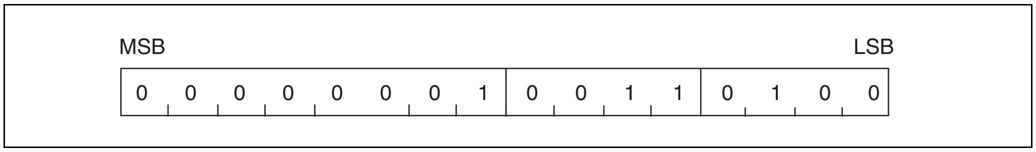
Execution cycles: 2+ a cycles

Instruction format:

	MSB											LSB
(n+0)	1	0	0	1	1	1	1	1	1	1	0	0	u4
(n+2)	CC								CRj				CRi

Example: COPOP #15, #1, CR3, CR4

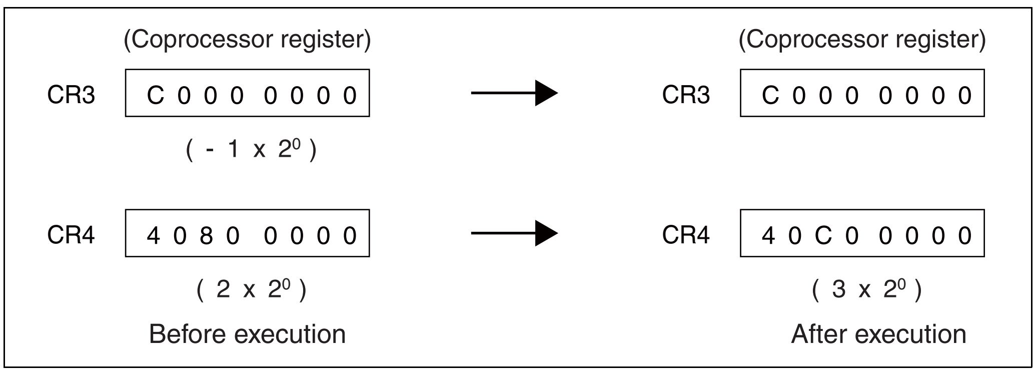
16-bit data is transferred through the bus to the coprocessor indicated by channel number 15.



Assuming that the coprocessor indicated by channel 15 is a single-precision floating-decimal calculation unit, the coprocessor command "CC" set as shown in Table 7.116-1 will have the following effect on coprocessor operation.

Table 7.116-1 Conditions for Coprocessor Command "CC" (COPOP)

CC	Calculation
00	Addition CRi + CRj → CRi
01	Subtraction CRi – CRj → CRi
02	Multiplication CRi × CRj → CRi
03	Division CRi ÷ CRj → CRi
Other	No operation



7.117 COPLD (Load 32-bit Data from Register to Coprocessor Register)

Transfers the 16-bit data consisting of parameters "CC", "Rj", "CRi" to the coprocessor indicated by channel number "u4", then on the next cycle transfers the contents of CPU general-purpose register "Rj" to that coprocessor.
Basically, this operation transfers data to a register within the coprocessor. The 32-bit data stored in CPU general-purpose register "Rj" is transferred to coprocessor register "CRi". Note that the actual interpretation of the fields "CC", "Rj", "CRi" is done by the coprocessor so that the detailed actual operation is determined by the specifications of the coprocessor.
If the coprocessor designated by the value "u4" is not mounted, a "coprocessor not found" trap is generated.
If the coprocessor designated by the value "u4" has generated an error in a previous operation, a "coprocessor error" trap is generated.

■ COPLD (Load 32-bit Data from Register to Coprocessor Register)

Assembler format: COPLD #u4, #CC, Rj, CRi

Operation: CC, Rj, CRi → Coprocessor on channel u4
Rj → CRi

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

Execution cycles: 1 + 2a cycles

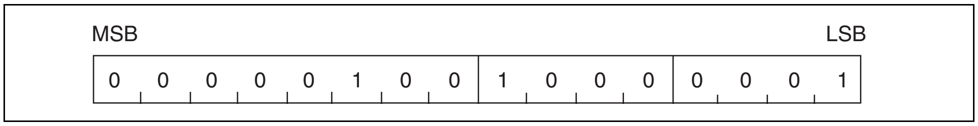
Instruction format:

	MSB												LSB
(n+0)	1	0	0	1	1	1	1	1	1	1	0	1	u4

(n+2)	CC	Rj	CRi

Example: COPLD #15, #4, R8, CR1

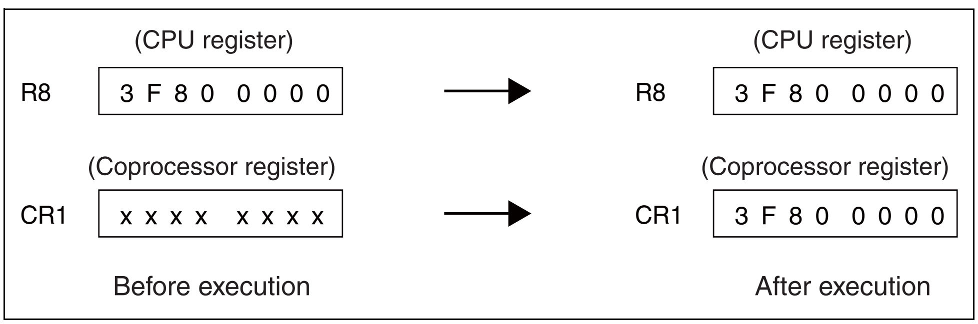
16-bit data is transferred through the bus to the coprocessor indicated by channel number 15. Next, the contents of general-purpose register "R8" are transferred through the bus to that coprocessor.



Assuming that the coprocessor indicated by channel 15 is a single-precision floating-decimal calculation unit, the coprocessor command "CC" set as shown in Table 7.117-1 will have the following effect on coprocessor operation.

Table 7.117-1 Conditions for Coprocessor Command "CC" (COPLD)

CC	Calculation
00	Addition CRi + CRj → CRi
01	Subtraction CRi – CRj → CRi
02	Multiplication CRi × CRj → CRi
03	Division CRi ÷ CRj → CRi
Other	No calculation



7.118 COPST (Store 32-bit Data from Coprocessor Register to Register)

Transfers the 16-bit data consisting of parameters "CC", "CRj", "Ri" to the coprocessor indicated by channel number "u4", then on the next cycle loads the data output by the coprocessor into CPU general-purpose register "Ri".
Basically, this operation transfers data from a register within the coprocessor. The 32-bit data stored in coprocessor register "CRj" is transferred to CPU general-purpose register "Ri". Note that the actual interpretation of the fields "CC", "CRj", "Ri" is done by the coprocessor so that the detailed actual operation is determined by the specifications of the coprocessor.
If the coprocessor designated by the value "u4" is not mounted, a "coprocessor not found" trap is generated.
If the coprocessor designated by the value "u4" has generated an error in a previous operation, a "coprocessor error" trap is generated.

■ COPST (Store 32-bit Data from Coprocessor Register to Register)

Assembler format: COPST #u4, #CC, CRj, Ri

Operation: CC, CRj, Ri → Coprocessor on channel u4
CRj → Ri

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

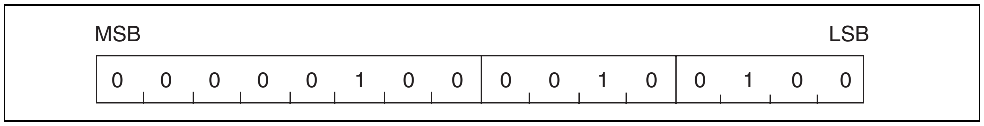
Execution cycles: 1 + 2a cycles

Instruction format:

	MSB												LSB
(n+0)	1	0	0	1	1	1	1	1	1	1	1	0	u4
(n+2)	CC								CRj				Ri

Example: COPST #15, #4, CR2, R4

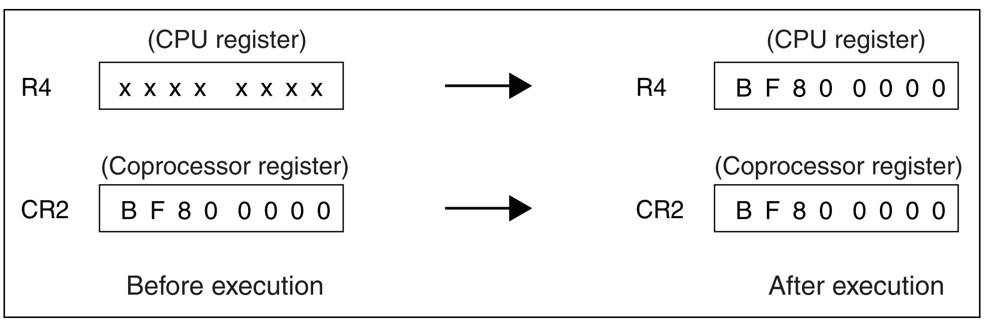
16-bit data is transferred through the bus to the coprocessor indicated by channel number 15. Next, the output data of the coprocessor are transferred through the bus to that coprocessor.



Assuming that the coprocessor indicated by channel 15 is a single-precision floating-decimal calculation unit, the coprocessor command "CC" set as shown in Table 7.118-1 will have the following effect on coprocessor operation.

Table 7.118-1 Conditions for Coprocessor Command "CC" (COPST)

CC	Calculation
00	Addition CRi + CRj → CRi
01	Subtraction CRi – CRj → CRi
02	Multiplication CRi × CRj → CRi
03	Division CRi ÷ CRj → CRi
Other	No calculation



7.119 COPSV (Save 32-bit Data from Coprocessor Register to Register)

Transfers the 16-bit data consisting of parameters "CC", "CRj", "Ri" to the coprocessor indicated by channel number u4, then on the next cycle loads the data output by the coprocessor to CPU general-purpose register "Ri".
Basically, this operation transfers data from a register within the coprocessor. The 32-bit data stored in coprocessor register "CRj" is transferred to CPU general-purpose register "Ri". Note that the actual interpretation of the fields "CC", "CRj", "Ri" is done by the coprocessor so that the detailed actual operation is determined by the specifications of the coprocessor.
If the coprocessor designated by the value "u4" is not mounted, a "coprocessor not found" trap is generated.
However, no "coprocessor error" trap will be generated even if the coprocessor designated by the value "u4" has generated an error in a previous operation.
The operation of this instruction is basically identical to "COPST", except for the above difference in the operation of the error trap.

■ COPSV (Save 32-bit Data from Coprocessor Register to Register)

Assembler format: COPSV #u4, #CC, CRj, Ri

Operation: CC, CRj, Ri → Coprocessor on channel u4
CRj → Ri

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

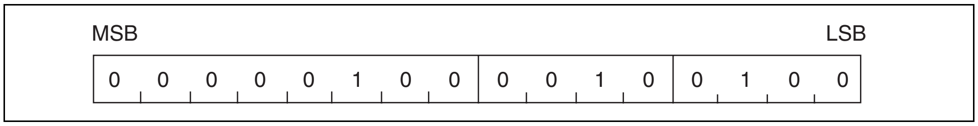
Execution cycles: 1 + 2a cycles

Instruction format:

	MSB											LSB
(n+0)	1	0	0	1	1	1	1	1	1	1	1	1	u4
(n+2)	CC								CRj			Ri

Example: COPSV #15, #4, CR2, R4

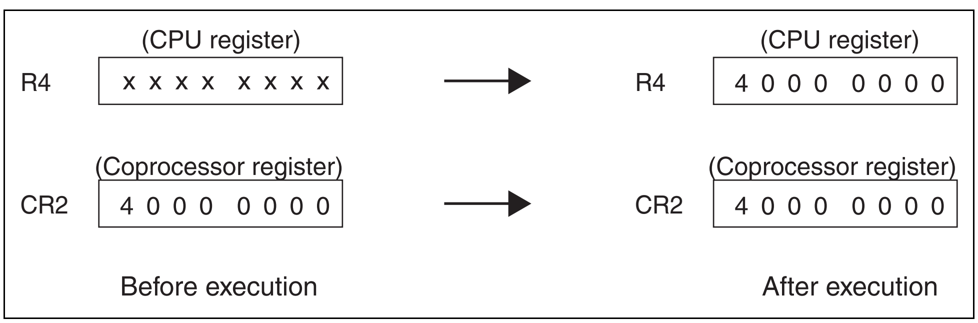
16-bit data is transferred through the bus to the coprocessor indicated by channel number 15. Next, the data output by the coprocessor is loaded into the CPU through the data bus. Note that no "coprocessor error" trap will be generated even if the coprocessor designated by the value "u4" has generated an error in a previous operation.



Assuming that the coprocessor indicated by channel 15 is a single-precision floating-decimal calculation unit, the coprocessor command "CC" set as shown in Table 7.119-1 will have the following effect on coprocessor operation.

Table 7.119-1 Conditions for Coprocessor Command "CC" (COPSV)

CC	Calculation
00	Addition CRi + CRj → CRi
01	Subtraction CRi – CRj → CRi
02	Multiplication CRi × CRj → CRi
03	Division CRi ÷ CRj → CRi
Other	No calculation



7.120 NOP (No Operation)

This instruction performs no operation.

■ NOP (No Operation)

Assembler format: NOP

Operation: This instruction performs no operation.

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

Execution cycles: 1 cycle

Instruction format:

MSB ... LSB

1	0	0	1	1	1	1	1	1	0	1	0	0	0	0	0

Example: NOP

Instruction bit pattern : 1001 1111 1010 0000

	Before execution			After execution
PC	8 3 4 3 8 3 4 A	→	PC	8 3 4 3 8 3 4 C

7.121 ANDCCR (And Condition Code Register and Immediate Data)

Takes the logical AND of the byte data in the condition code register (CCR) and the immediate data, and returns the results into the "CCR".

■ ANDCCR (And Condition Code Register and Immediate Data)

Assembler format: ANDCCR #u8

Operation: CCR and u8 → CCR

Flag change:

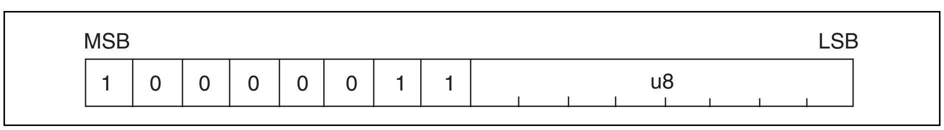
S	I	N	Z	V	C
C	C	C	C	C	C

S, I, N, Z, V, and C: Varies according to results of calculation.

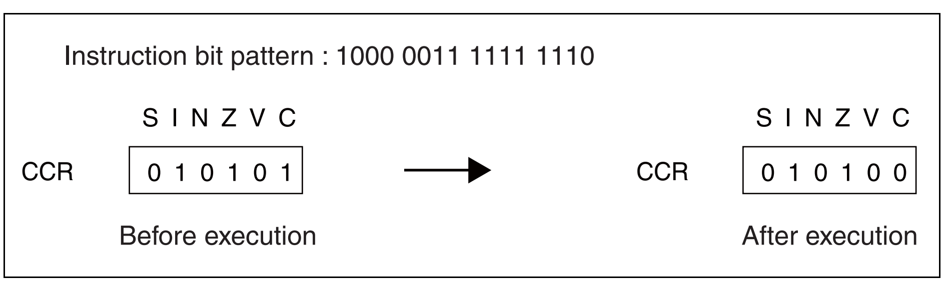
Execution cycles: c cycle(s)

The number of execution cycles is normally "1". However, if the instruction immediately after involves read or write access to memory address "R15", the system stack pointer (SSP) or the user stack pointer (USP), then an interlock is applied and the value becomes 2 cycles.

Instruction format:



Example: ANDCCR #0FEH



7.122 ORCCR (Or Condition Code Register and Immediate Data)

Takes the logical OR of the byte data in the condition code register (CCR) and the immediate data, and returns the results into the "CCR".

■ ORCCR (Or Condition Code Register and Immediate Data)

Assembler format: ORCCR #u8

Operation: CCR or u8 → CCR

Flag change:

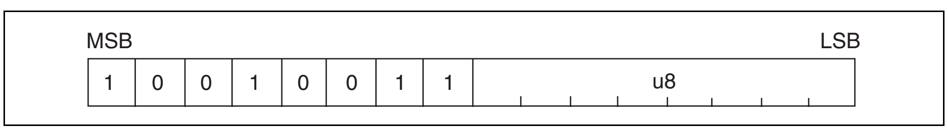
S	I	N	Z	V	C
C	C	C	C	C	C

S, I, N, Z, V, and C: Varies according to results of calculation.

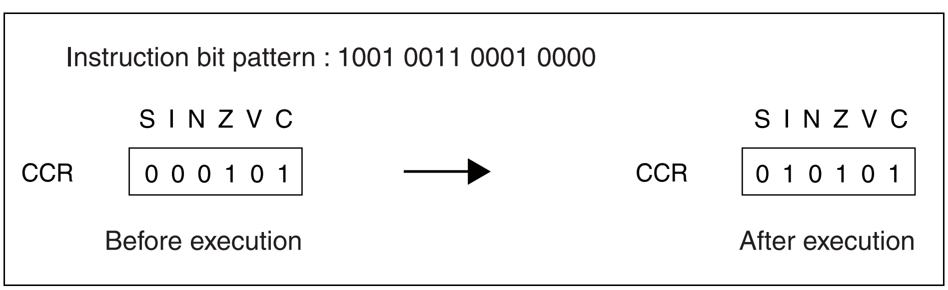
Execution cycles: c cycle(s)

The number of execution cycles is normally "1". However, if the instruction immediately after involves read or write access to memory address "R15", the system stack pointer (SSP) or the user stack pointer (USP), then an interlock is applied and the value becomes 2 cycles.

Instruction format:



Example: ORCCR #10H



7.123 STILM (Set Immediate Data to Interrupt Level Mask Register)

Transfers the immediate data to the interrupt level mask register (ILM) in the program status (PS).
Only the lower 5 bits (bit4 to bit0) of the immediate data are valid.
At the time this instruction is executed, if the value of the interrupt level mask register (ILM) is in the range 16 to 31, only new "ILM" settings between 16 and 31 can be entered. If the value "u8" is in the range 0 to 15, the value 16 will be added to that data before being transferred to the "ILM". If the original "ILM" value is in the range 0 to 15, then any value between 0 and 31 can be transferred to the "ILM".

■ STILM (Set Immediate Data to Interrupt Level Mask Register)

Assembler format: STILM #u8

Operation: u8 → ILM

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

Execution cycles: 1 cycle

Instruction format:

MSB ... LSB

1	0	0	0	0	1	1	1	u8

Example: STILM #14H

Instruction bit pattern : 1000 0111 0001 0100

ILM 1 1 1 1 1 → ILM 1 0 1 0 0

Before execution — After execution

7.124 ADDSP (Add Stack Pointer and Immediate Data)

Adds 4 times the immediate data as a signed extended value, to the value in "R15".

■ ADDSP (Add Stack Pointer and Immediate Data)

Assembler format: ADDSP #s10

Operation: R15 + exts (s8 × 4) → R15

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

Execution cycles: 1 cycle

Instruction format:

MSB … LSB

1	0	1	0	0	0	1	1	s8

Example: ADDSP # – 4

Instruction bit pattern : 1010 0011 1111 1111

	Before execution			After execution
R15	8 0 0 0 0 0 0 0	→	R15	7 F F F F F F C

7.125 EXTSB (Sign Extend from Byte Data to Word Data)

Extends the byte data indicated by "Ri" to word data as a signed binary value.

■ EXTSB (Sign Extend from Byte Data to Word Data)

Assembler format: EXTSB Ri

Operation: exts (Ri) → Ri (byte → word)

Flag change:

N	Z	V	C
–	–	–	–

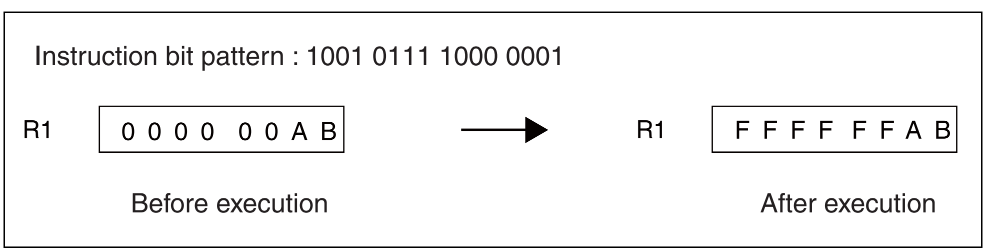
N, Z, V, and C: Unchanged

Execution cycles: 1 cycle

Instruction format:

MSB											LSB
1	0	0	1	0	1	1	1	1	0	0	Ri

Example: EXTSB R1



7.126 EXTUB (Unsign Extend from Byte Data to Word Data)

Extends the byte data indicated by "Ri" to word data as an unsigned binary value.

■ EXTUB (Unsign Extend from Byte Data to Word Data)

Assembler format: EXTUB Ri

Operation: extu (Ri) → Ri (byte → word)

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

Execution cycles: 1 cycle

Instruction format:

MSB ... LSB

1	0	0	1	0	1	1	1	1	0	0	1	Ri

Example: EXTUB R1

Instruction bit pattern : 1001 0111 1001 0001

	Before execution			After execution
R1	F F F F F F F F	→	R1	0 0 0 0 0 0 F F

7.127 EXTSH (Sign Extend from Byte Data to Word Data)

Extends the half-word data indicated by "Ri" to word data as a signed binary value.

■ EXTSH (Sign Extend from Byte Data to Word Data)

Assembler format: EXTSH Ri

Operation: exts (Ri) → Ri (half-word → word)

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

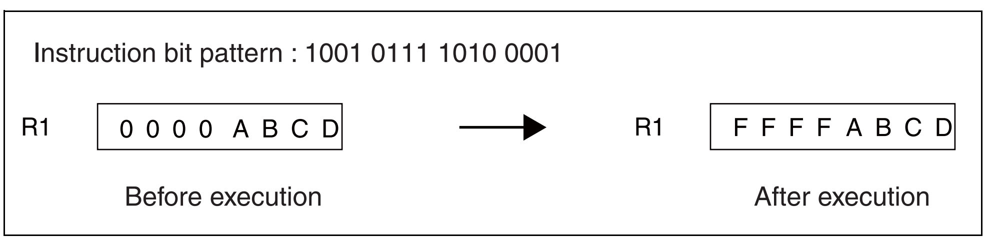
Execution cycles: 1 cycle

Instruction format:

MSB (left) … LSB (right)

1	0	0	1	0	1	1	1	1	0	1	0	Ri

Example: EXTSH R1



7.128 EXTUH (Unsigned Extend from Byte Data to Word Data)

Extends the half-word data indicated by "Ri" to word data as an unsigned binary value.

■ EXTUH (Unsigned Extend from Byte Data to Word Data)

Assembler format: EXTUH Ri

Operation: extu (Ri) → Ri (half-word → word)

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

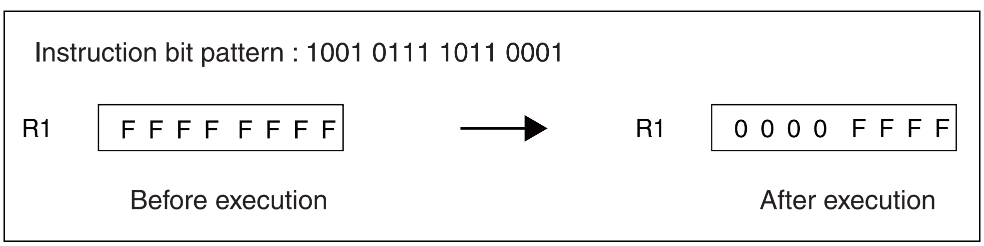
Execution cycles: 1 cycle

Instruction format:

MSB → LSB

1	0	0	1	0	1	1	1	1	0	1	1	Ri

Example: EXTUH R1



7.129 LDM0 (Load Multiple Registers)

The "LDM0" instruction accepts registers in the range R0 to R7 as members of the parameter "reglist". (See Table 7.129-1.)
Registers are processed in ascending numerical order.

■ LDM0 (Load Multiple Registers)

Assembler format: LDM0 (reglist)

Operation: The following operations are repeated according to the number of registers specified in the parameter "reglist".

(R15) → Ri

R15 + 4 → R15

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

Execution cycles: If "n" is the number of registers specified in the parameter "reglist", the execution cycles required are as follows.

If n=0: 1 cycle

For other values of n: a (n – 1) + b + 1 cycles

Instruction format:

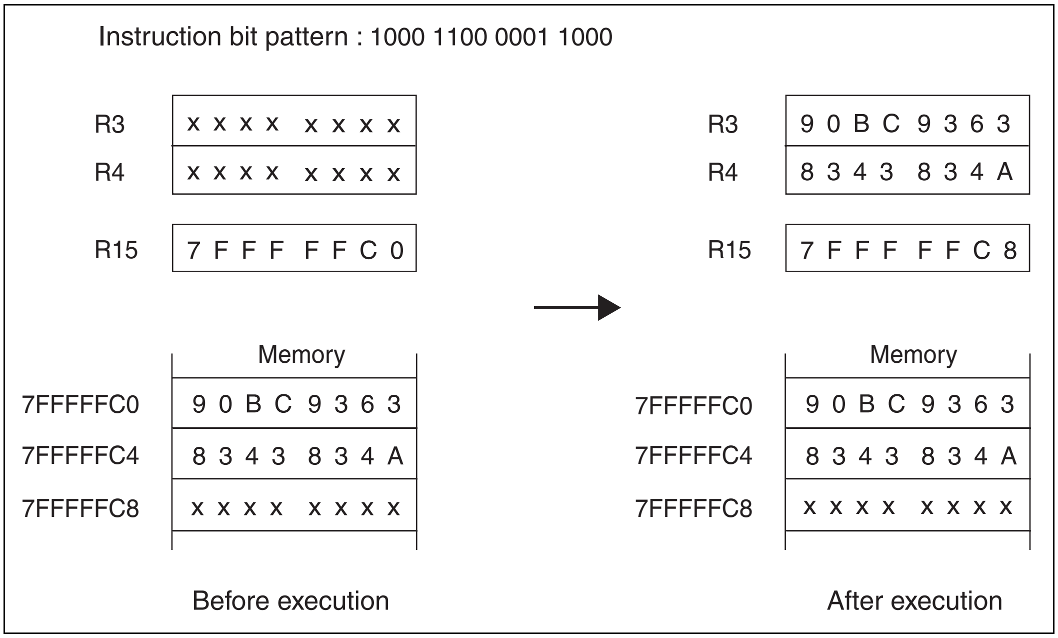
MSB ... LSB

1	0	0	0	1	1	0	0	reglist

Table 7.129-1 Bit Values and Register Numbers for "reglist" (LDM0)

Bit	Register	Bit	Register
7	R7	3	R3
6	R6	2	R2
5	R5	1	R1
4	R4	0	R0

Example: LDM0 (R3, R4)



7.130 LDM1 (Load Multiple Registers)

The LDM1 instruction accepts registers in the range R8 to R15 as members of the parameter "reglist" (See Table 7.130-1.).
Registers are processed in ascending numerical order.
If "R15" is specified in the parameter "reglist", the final contents of "R15" will be read from memory.

■ LDM1 (Load Multiple Registers)

Assembler format: LDM1 (reglist)

Operation: The following operations are repeated according to the number of registers specified in the parameter "reglist".

(R15) → Ri

R15 + 4 → R15

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

Execution cycles: If "n" is the number of registers specified in the parameter "reglist", the execution cycles required are as follows.

If n=0: 1 cycle

For other values of n: a (n – 1) + b + 1 cycles

Instruction format:

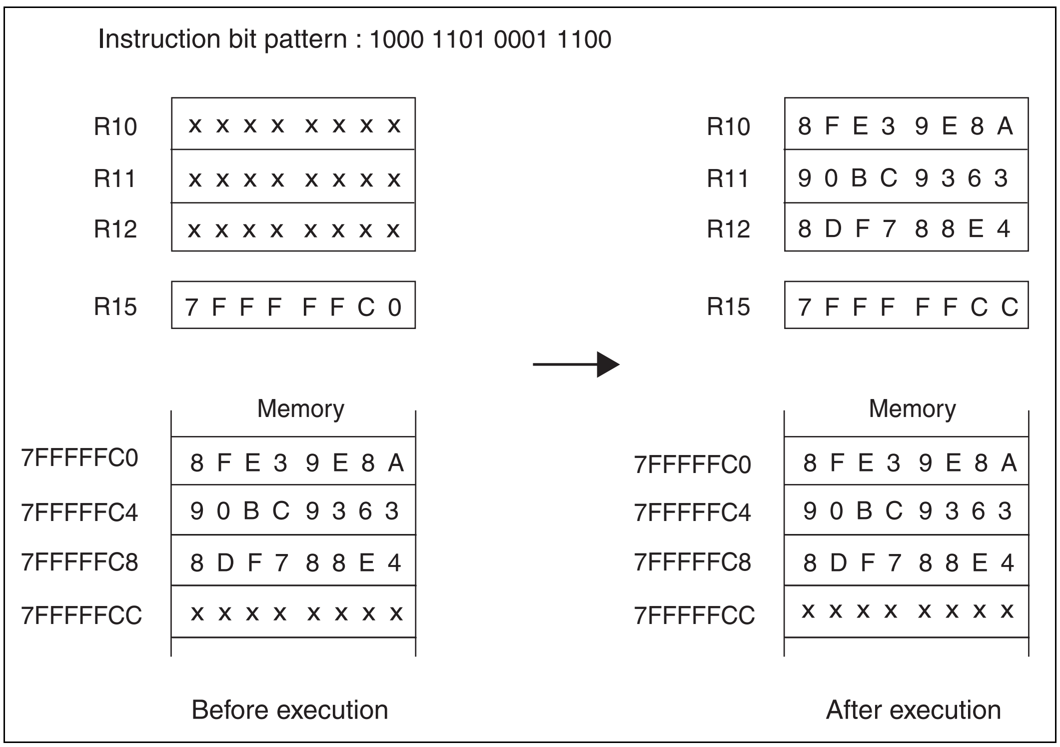
MSB ... LSB

1	0	0	0	1	1	0	1	reglist

Table 7.130-1 Bit Values and Register Numbers for "reglist" (LDM1)

Bit	Register	Bit	Register
7	R15	3	R11
6	R14	2	R10
5	R13	1	R9
4	R12	0	R8

Example: LDM1 (R10, R11, R12)



7.131 STM0 (Store Multiple Registers)

The "STM0" instruction accepts registers in the range R0 to R7 as members of the parameter "reglist" (See Table 7.131-1.) .
Registers are processed in descending numerical order.

■ STM0 (Store Multiple Registers)

Assembler format: STM0 (reglist)

Operation: The following operations are repeated according to the number of registers specified in the parameter "reglist".

$R15 - 4 \rightarrow R15$

$Ri \rightarrow (R15)$

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

Execution cycles: If "n" is the number of registers specified in the parameter "reglist", the execution cycles required are as follows.

$a \times n + 1$ cycle

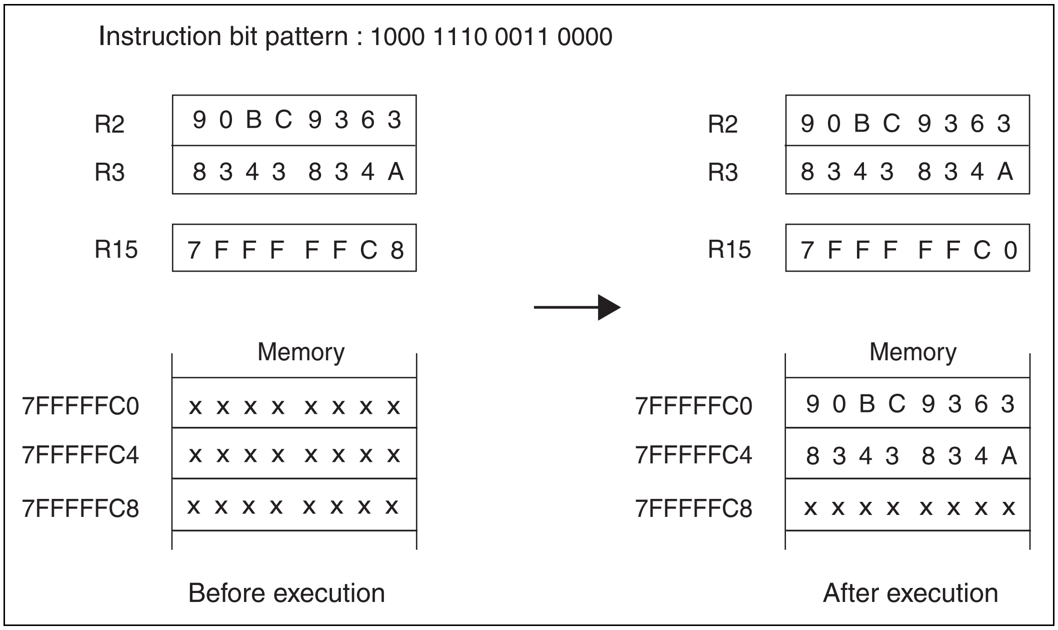
Instruction format:

MSB								LSB
1	0	0	0	1	1	1	0	reglist

Table 7.131-1 Bit Values and Register Numbers for "reglist" (STM0)

Bit	Register	Bit	Register
7	R0	3	R4
6	R1	2	R5
5	R2	1	R6
4	R3	0	R7

Example: STM0 (R2, R3)



7.132 STM1 (Store Multiple Registers)

The "STM1" instruction accepts registers in the range R8 to R15 as members of the parameter "reglist" (See Table 7.132-1.).
Registers are processed in descending numerical order.
If "R15" is specified in the parameter "reglist", the contents of "R15" retained before the instruction is executed will be written to memory.

■ STM1 (Store Multiple Registers)

Assembler format: STM1 (reglist)

Operation: The following operations are repeated according to the number of registers specified in the parameter "reglist".

R15 – 4 → R15

Ri → (R15)

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

Execution cycles: If "n" is the number of registers specified in the parameter "reglist", the execution cycles required are as follows.

a × n + 1 cycles

Instruction format:

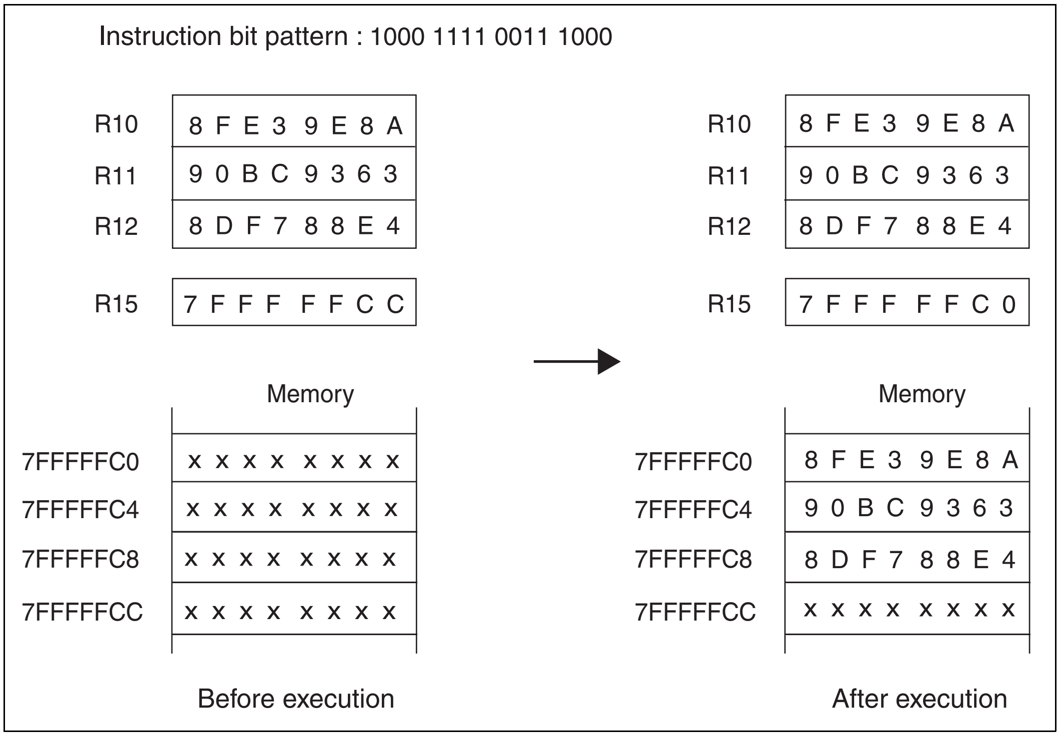
MSB … LSB

1	0	0	0	1	1	1	1	reglist

Table 7.132-1 Bit Values and Register Numbers for "reglist" (STM1)

Bit	Register	Bit	Register
7	R8	3	R12
6	R9	2	R13
5	R10	1	R14
4	R11	0	R15

Example: STM1 (R10, R11, R12)



7.133 ENTER (Enter Function)

This instruction is used for stack frame generation processing for high level languages. The value "u8" is calculated as an unsigned value.

■ ENTER (Enter Function)

Assembler format: ENTER #u10

Operation: R14 → (R15 – 4)
R15 – 4 → R14
R15 – extu (u8 × 4) → R15

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

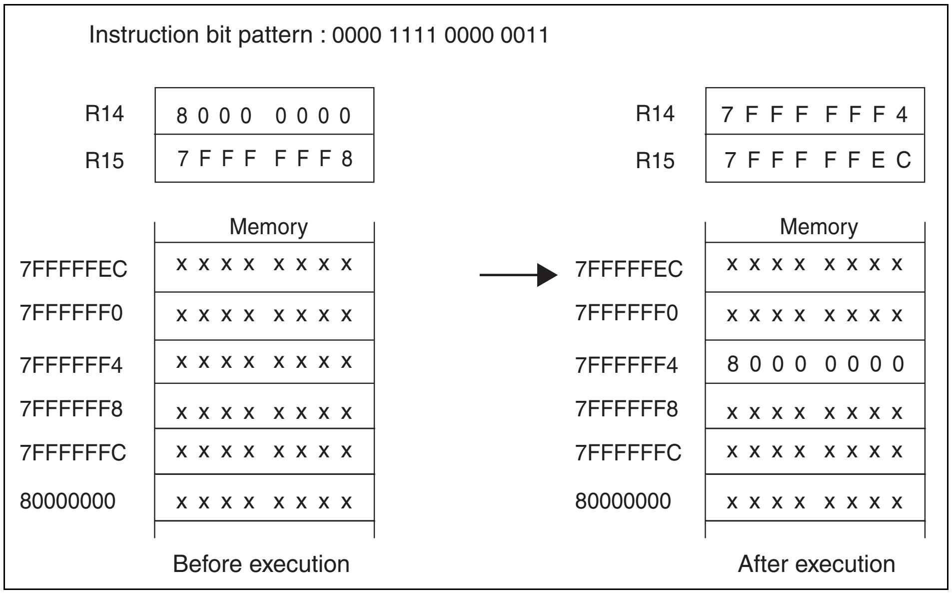
Execution cycles: 1 + a cycles

Instruction format:

MSB ... LSB

0	0	0	0	1	1	1	1	u8

Example: ENTER #0CH



7.134 LEAVE (Leave Function)

This instruction is used for stack frame release processing for high level languages.

■ LEAVE (Leave Function)

Assembler format: LEAVE

Operation: R14 + 4 → R15
(R15 – 4) → R14

Flag change:

N	Z	V	C
–	–	–	–

N, Z, V, and C: Unchanged

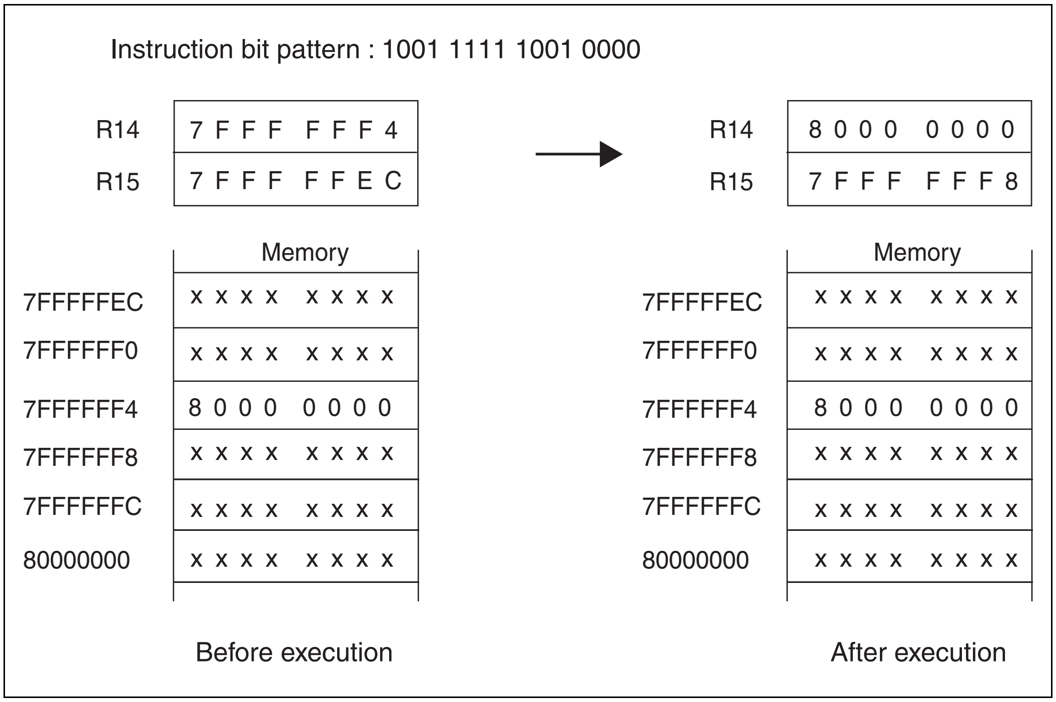
Execution cycles: b cycle(s)

Instruction format:

MSB ... LSB

1	0	0	1	1	1	1	1	1	0	0	1	0	0	0	0

Example: LEAVE



7.135 XCHB (Exchange Byte Data)

Exchanges the contents of the byte address indicated by "Rj" and those indicated by "Ri".
The lower 8 bits of data originally at "Ri" are transferred to the byte address indicated by "Rj", and the data originally at "Rj" is extended with zeros and transferred to "Ri".
The CPU will not accept hold requests between the memory read operation and the memory write operation of this instruction.

■ XCHB (Exchange Byte Data)

Assembler format: XCHB @Rj, Ri

Operation:
Ri → TEMP
extu ((Rj)) → Ri
TEMP → (Rj)

Flag change:

N	Z	V	C
–	–	–	–

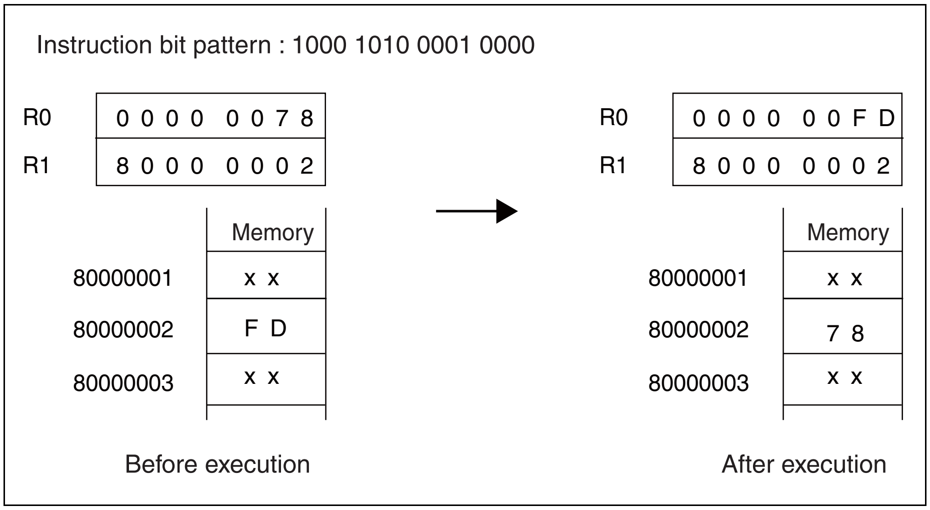
N, Z, V, and C: Unchanged

Execution cycles: 2a cycles

Instruction format:

MSB									LSB
1	0	0	0	1	0	1	0	Rj	Ri

Example: XCHB @R1, R0



APPENDIX

The appendix section includes lists of CPU instructions used in the FR family, as well as instruction map diagrams.

APPENDIX A Instruction Lists

Appendix A includes a description of symbols used in instruction lists, plus the instruction lists.

A.1 Symbols Used in Instruction Lists

This section describes symbols used in the FR family instruction lists.

■ Symbols Used in Instruction Lists

● Symbols in Mnemonic and Operation Columns

- i4 .... 4-bit immediate data, range 0 to 15 with zero extension, and –16 to –1 with minus extension
- i8 .... unsigned 8-bit immediate data, range 0 to 255
- i20 .... unsigned 20-bit immediate data, range 00000_H to $FFFFF_H$
- i32 .... unsigned 32-bit immediate data, range 00000000_H to $FFFFFFFF_H$
- s8............ signed 8-bit immediate data, range –128 to 127
- s10.......... signed 10-bit immediate data, range –512 to 508 (in multiples of 4)
- u4 .... unsigned 4-bit immediate data, range 0 to 15
- u8 .... unsigned 8-bit immediate data, range 0 to 255
- u10 .... unsigned 10-bit immediate data, range 0 to 1020 (multiples of 4)
- udisp6...... unsigned 6-bit address values, range 0 to 60 (multiples of 4)
- disp8........ signed 8-bit address values, range –0x80 to 0x7F
- disp9........ signed 9-bit address values, range –0x100 to 0xFE (multiples of 2)
- disp10...... signed 10-bit address values, range –0x200 to 0x1FC (multiples of 4)
- dir8.......... unsigned 8-bit address values, range 0 to 0xFF
- dir9.......... unsigned 9-bit address values, range 0 to 0x1FE (multiples of 2)
- dir10........ unsigned 10-bit address values, range 0 to 0x3FC(multiples of 4)
- label9....... signed 9-bit branch address, range –0x100 to 0xFE (multiples of 2) for the value of PC
- label12..... signed 12-bit branch address, range –0x800 to 0x7FE (multiples of 2) for the value of PC
- Ri, Rj....... indicates a general-purpose register (R00 to R15)
- Rs .... indicates a dedicated register (TBR, RP, USP, SSP, MDH, MDL)

● Symbols in Operation Column

- extu()....... indicates a zero extension operation, in which values lacking higher bits are complemented by adding the value "0" as necessary.
- extn()....... indicates a minus extension operation, in which values lacking higher bits are complemented by adding the value "1" as necessary.
- exts() .... indicates a sign extension operation in which a zero extension is performed for the data within ( ) in which the MSB is 0 and a minus extension is performed for the data in which the MSB is 1.
- ( )............ indicates indirect addressing, which values reading or loading from/to the memory address where the registers within ( ) or the formula indicate.
- { }........... indicates the calculation priority; ( ) is used for specifying indirect address

● Format Column

A to F .... format TYPE-A through F as described in Section "6.1 Instruction Formats".

● OP Column

"OP" codes have the following significance according to the format type listed in the format column.

- Format types A, C, D ..... 2-digit hexadecimal value represents 8-bit "OP" code.
- Format type B .... 2-digit hexadecimal value represents higher 4 bits of "OP" code, lower 4 bits "0".
- Format type E .... 4-digit hexadecimal value with higher 2 digits representing higher 8-bits of "OP" code, next digit representing 4-bit "SUB-OP" code, last digit "0".
- Format type F................. 2-digit hexadecimal code representing higher 5 bits of "OP" code, remainder "0".

● Cycle (CYC) Column

Numerical values represent machine cycles, variables "a" through "d" have a minimum value of 1.

- a............... Memory access cycles, may be increased by "Ready" function.
- b .... Memory access cycles, may be increased by "Ready" function. Note that if the next instruction references a register involved in a "LD" operation an interlock will be applied, increasing the number of execution cycles from 1 cycle to 2 cycles.
- c............... If the instruction immediately after is a read or write operation involving register "R15", or the "SSP" or "USP" pointers, or the instruction format is TYPE-A, an interlock will be applied, increasing the number of execution cycles from 1 cycle to 2 cycles.
- d .... If the instruction immediately after references the "MDH/MDL" register, interlock will be applied, increasing the number of execution cycles from 1 cycle to 2 cycles.
  When dedicated register such as TBR, RP, USP, SSP, MDH, and MDL is accessed with ST Rs, @-R15 command just after DIV1 command, an interlock is always brought, increasing the number of execution cycles from 1 cycle to 2 cycles.

● FLAG Column

- C.............. varies according to results of operation.
- – .... no change
- 0 .... value becomes "0".
- 1 .... value becomes "1".

A.2 Instruction Lists

The full instruction set of the FR family CPU is 165 instructions, consisting of the following sixteen types. These instructions are listed in Table A.2-1 through Table A.2-16.

- **Add/Subtract Instructions (10 Instructions)**
- **Compare Instructions (3 Instructions)**
- **Logical Calculation Instructions (12 Instructions)**
- **Bit Operation Instructions (8 Instructions)**
- **Multiply/Divide Instructions (10 Instructions)**
- **Shift Instructions (9 Instructions)**
- **Immediate Data Transfer Instructions (3 Instructions)**
- **Memory Load Instructions (13 Instructions)**
- **Memory Store Instructions (13 Instructions)**
- **Inter-register Transfer Instructions / Dedicated Register Transfer Instructions (5 Instructions)**
- **Non-delayed Branching Instructions (23 Instructions)**
- **Delayed Branching Instructions (20 Instructions)**
- **Direct Addressing Instructions (14 Instructions)**
- **Resource Instructions (2 Instructions)**
- **Coprocessor Instructions (4 Instructions)**
- **Other Instructions (16 Instructions)**

■ Instruction Lists

Table A.2-1 Add/Subtract Instructions (10 Instructions)

Mnemonic	Format	OP	CYC	FLAG NZVC	Operation	Remarks
ADD Rj, Ri	A	A6	1	CCCC	Ri + Rj → Ri	
ADD #i4, Ri	C	A4	1	CCCC	Ri + extu(i4) → Ri	Zero extension
ADD2 #i4, Ri	C	A5	1	CCCC	Ri + extn(i4) → Ri	Minus extension
ADDC Rj, Ri	A	A7	1	CCCC	Ri + Rj + c → Ri	Add with carry
ADDN Rj, Ri	A	A2	1	– – – –	Ri + Rj → Ri	
ADDN #i4, Ri	C	A0	1	– – – –	Ri + extu(i4) → Ri	Zero extension
ADDN2 #i4, Ri	C	A1	1	– – – –	Ri + extn(i4) → Ri	Minus extension
SUB Rj, Ri	A	AC	1	CCCC	Ri – Rj → Ri	
SUBC Rj, Ri	A	AD	1	CCCC	Ri – Rj – c → Ri	Subtract with carry
SUBN Rj, Ri	A	AE	1	– – – –	Ri – Rj → Ri	

Table A.2-2 Compare Instructions (3 Instructions)

Mnemonic	Format	OP	CYC	FLAG NZVC	Operation	Remarks
CMP Rj, Ri	A	AA	1	CCCC	Ri – Rj	
CMP #i4, Ri	C	A8	1	CCCC	Ri – extu(i4)	Zero extension
CMP2 #i4, Ri	C	A9	1	CCCC	Ri – extn(i4)	Minus extension

Table A.2-3 Logical Calculation Instructions (12 Instructions)

Mnemonic	Format	OP	CYC	FLAG NZVC	Operation	RMW	Remarks
AND Rj, Ri	A	82	1	CC – –	Ri &= Rj	-	Word
AND Rj, @Ri	A	84	1+2a	CC – –	(Ri) &= Rj	❍	Word
ANDH Rj, @Ri	A	85	1+2a	CC – –	(Ri) &= Rj	❍	Half-word
ANDB Rj, @Ri	A	86	1+2a	CC – –	(Ri) &= Rj	❍	Byte
OR Rj, Ri	A	92	1	CC – –	Ri |= Rj	-	Word
OR Rj, @Ri	A	94	1+2a	CC – –	(Ri) |= Rj	❍	Word
ORH Rj, @Ri	A	95	1+2a	CC – –	(Ri) |= Rj	❍	Half-word
ORB Rj, @Ri	A	96	1+2a	CC – –	(Ri) |= Rj	❍	Byte
EOR Rj, Ri	A	9A	1	CC – –	Ri ^= Rj	-	Word
EOR Rj, @Ri	A	9C	1+2a	CC – –	(Ri) ^= Rj	❍	Word
EORH Rj, @Ri	A	9D	1+2a	CC – –	(Ri) ^= Rj	❍	Half-word
EORB Rj, @Ri	A	9E	1+2a	CC – –	(Ri) ^= Rj	❍	Byte

Table A.2-4 Bit Operation Instructions (8 Instructions)

Mnemonic	Format	OP	CYC	FLAG NZVC	Operation	RMW	Remarks
BANDL #u4, @Ri (u4: 0 to $0F_H$)	C	80	1+2a	– – – –	(Ri)&={$F0_H$+u4}	❍	Lower 4-bit operation
BANDH #u4, @Ri (u4: 0 to $0F_H$)	C	81	1+2a	– – – –	(Ri)&={{u4<<4}+F_H}	❍	Higher 4-bit operation
BORL #u4, @Ri (u4: 0 to $0F_H$)	C	90	1+2a	– – – –	(Ri) | = u4	❍	Lower 4-bit operation
BORH #u4, @Ri (u4: 0 to $0F_H$)	C	91	1+2a	– – – –	(Ri) | = {u4<<4}	❍	Higher 4-bit operation
BEORL #u4, @Ri (u4: 0 to $0F_H$)	C	98	1+2a	– – – –	(Ri) ^ = u4	❍	Lower 4-bit operation
BEORH #u4, @Ri (u4: 0 to $0F_H$)	C	99	1+2a	– – – –	(Ri) ^ = {u4<<4}	❍	Higher 4-bit operation
BTSTL #u4, @Ri (u4: 0 to $0F_H$)	C	88	2+a	0C – –	(Ri) & u4	-	Lower 4-bit test
BTSTH #u4, @Ri (u4: 0 to $0F_H$)	C	89	2+a	CC – –	(Ri) & {u4<<4}	-	Higher 4-bit test

Table A.2-5 Multiply/Divide Instructions (10 Instructions)

Mnemonic	Format	OP	CYC	FLAG NZVC	Operation	Remarks
MUL Rj,Ri	A	AF	5	CCC –	Rj × Ri → MDH,MDL	32bits × 32bits=64bits
MULU Rj,Ri	A	AB	5	CCC –	Rj × Ri → MDH,MDL	Unsigned
MULH Rj,Ri	A	BF	3	CC – –	Rj × Ri → MDL	16bits × 16bits=32bits
MULUH Rj,Ri	A	BB	3	CC – –	Rj × Ri → MDL	Unsigned
DIV0S Ri	E	97-4	1	– – – –		Step operation
DIV0U Ri	E	97-5	1	– – – –		32bits/32bits=32bits
DIV1 Ri	E	97-6	d	– C– C		
DIV2 Ri	E	97-7	1	– C– C		
DIV3	E	9F-6	1	– – – –		
DIV4S	E	9F-7	1	– – – –		

Table A.2-6 Shift Instructions (9 Instructions)

Mnemonic	Format	OP	CYC	FLAG NZVC	Operation	Remarks
LSL Rj, Ri	A	B6	1	CC – C	Ri << Rj → Ri	Logical shift
LSL #u4, Ri	C	B4	1	CC – C	Ri << u4 → Ri	
LSL2 #u4, Ri	C	B5	1	CC – C	Ri <<{u4+16} → Ri	
LSR Rj, Ri	A	B2	1	CC – C	Ri >> Rj → Ri	Logical shift
LSR #u4, Ri	C	B0	1	CC – C	Ri >> u4 → Ri	
LSR2 #u4, Ri	C	B1	1	CC – C	Ri >>{u4+16} → Ri	
ASR Rj, Ri	A	BA	1	CC – C	Ri >> Rj → Ri	Arithmetic shift
ASR #u4, Ri	C	B8	1	CC – C	Ri >> u4 → Ri	
ASR2 #u4, Ri	C	B9	1	CC – C	Ri >>{u4+16} → Ri	

Table A.2-7 Immediate Data Transfer Instructions (Immediate Transfer Instructions for Immediate Value Set or 16-bit or 32-bit Values) (3 Instructions)

Mnemonic	Format	OP	CYC	FLAG NZVC	Operation	Remarks
LDI:32 #i32, Ri	E	9F-8	3	– – – –	i32 → Ri	
LDI:20 #i20, Ri	C	9B	2	– – – –	i20 → Ri	Higher 12 bits are zeros
LDI:8 #i8, Ri	B	C0	1	– – – –	i8 → Ri	Higher 24 bits are zeros

Table A.2-8 Memory Load Instructions (13 Instructions)

Mnemonic	Format	OP	CYC	FLAG NZVC	Operation	Remarks
LD @Rj, Ri	A	04	b	– – – –	(Rj) → Ri	
LD @(R13,Rj), Ri	A	00	b	– – – –	(R13+Rj) → Ri	
LD @(R14,disp10), Ri	B	20	b	– – – –	(R14+disp10) → Ri	
LD @(R15,udisp6), Ri	C	03	b	– – – –	(R15+udisp6) → Ri	
LD @R15+, Ri	E	07-0	b	– – – –	(R15) → Ri,R15+=4	
LD @R15+, Rs	E	07-8	b	– – – –	(R15) → Rs, R15+=4	Rs: dedicated register
LD @R15+, PS	E	07-9	1+a+b	CCCC	(R15) → PS, R15+=4	
LDUH @Rj, Ri	A	05	b	– – – –	(Rj) → Ri	Zero extension
LDUH @(R13,Rj), Ri	A	01	b	– – – –	(R13+Rj) → Ri	Zero extension
LDUH @(R14,disp9), Ri	B	40	b	– – – –	(R14+disp9) → Rj	Zero extension
LDUB @Rj, Ri	A	06	b	– – – –	(Rj) → Ri	Zero extension
LDUB @(R13,Rj), Ri	A	02	b	– – – –	(R13+Rj) → Ri	Zero extension
LDUB @(R14,disp8), Ri	B	60	b	– – – –	(R14+disp8) → Ri	Zero extension

Note:

The field "o8" in the TYPE-B instruction format and the field "u4" in the TYPE-C format have the following relation to the values "disp8" to "disp10" in assembly notation.

- disp8 → o8=disp8
- disp9 → o8=disp9 >> 1
- disp10 → o8=disp10 >> 2
- udisp6 → u4=udisp6 >> 2

Table A.2-9 Memory Store Instructions (13 Instructions)

Mnemonic	Format	OP	CYC	FLAG NZVC	Operation	Remarks
ST Ri, @Rj	A	14	a	– – – –	Ri → (Rj)	Word
ST Ri, @(R13,Rj)	A	10	a	– – – –	Ri → (R13+Rj)	Word
ST Ri, @(R14,disp10)	B	30	a	– – – –	Ri → (R14+disp10)	Word
ST Ri, @(R15,udisp6)	C	13	a	– – – –	Ri → (R15+udisp6)	
ST Ri, @-R15	E	17-0	a	– – – –	R15–=4,Ri → (R15)	
ST Rs, @-R15	E	17-8	a	– – – –	R15–=4, Rs → (R15)	Rs: dedicated register
ST PS, @-R15	E	17-9	a	– – – –	R15–=4, PS → (R15)	
STH Ri, @Rj	A	15	a	– – – –	Ri → (Rj)	Half-word
STH Ri, @(R13,Rj)	A	11	a	– – – –	Ri → (R13+Rj)	Half-word
STH Ri, @(R14,disp9)	B	50	a	– – – –	Ri → (R14+disp9)	Half-word
STB Ri, @Rj	A	16	a	– – – –	Ri → (Rj)	Byte
STB Ri, @(R13,Rj)	A	12	a	– – – –	Ri → (R13+Rj)	Byte
STB Ri, @(R14,disp8)	B	70	a	– – – –	Ri → (R14+disp8)	Byte

Note:

The field "o8" in the TYPE-B instruction format and the field "u4" in the TYPE-C format have the following relation to the values "disp8" to "disp10" in assembly notation.

- disp8 → o8=disp8
- disp9 → o8=disp9 >> 1
- disp10 → o8=disp10 >> 2
- udisp6 → u4=udisp6 >> 2

Table A.2-10 Inter-register Transfer Instructions / Dedicated Register Transfer Instructions (5 Instructions)

Mnemonic	Format	OP	CYC	FLAG NZVC	Operation	Remarks
MOV Rj, Ri	A	8B	1	– – – –	Rj → Ri	Transfer between general-purpose registers
MOV Rs, Ri	A	B7	1	– – – –	Rs → Ri	Rs: dedicated register
MOV Ri, Rs	A	B3	1	– – – –	Ri → Rs	Rs: dedicated register
MOV PS, Ri	E	17-1	1	– – – –	PS → Ri	
MOV Ri, PS	E	07-1	c	CCCC	Ri → PS	

Table A.2-11 Non-delayed Branching Instructions (23 Instructions)

Mnemonic	Format	OP	CYC	FLAG NZVC	Operation	Remarks
JMP @Ri	E	97-0	2	– – – –	Ri → PC	
CALL label12 CALL @Ri	F E	D0 97-1	2 2	– – – – – – – –	PC+2 → RP ,PC+2+rel11×2 → PC PC+2 → RP, Ri → PC	
RET	E	97-2	2	– – – –	RP → PC	Return
INT #u8	D	1F	3+3a	– – – –	SSP–=4,PS → (SSP),SSP–=4,PC+2 → (SSP), 0 → I flag, 0 → S flag, (TBR+3FC–u8×4) → PC	
INTE	E	9F-3	3+3a	– – – –	SSP–=4,PS → (SSP),SSP–=4,PC+2 → (SSP), 0 → S flag, 4 → ILM, (TBR+3D8–u8×4) → PC	
RETI	E	97-3	2+2a	CCCC	(R15) → PC,R15+=4,(R15) → PS,R15+=4	
BNO label9 BRA label9 BEQ label9 BNE label9 BC label9 BNC label9 BN label9 BP label9 BV label9 BNV label9 BLT label9 BGE label9 BLE label9 BGT label9 BLS label9 BHI label9	D D D D D D D D D D D D D D D D	E1 E0 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF	1 2 2/1 2/1 2/1 2/1 2/1 2/1 2/1 2/1 2/1 2/1 2/1 2/1 2/1 2/1	– –	No branch PC+2+rel8×2 → PC PC+2+rel8×2 → PC if Z==1 PC+2+rel8×2 → PC if Z==0 PC+2+rel8×2 → PC if C==1 PC+2+rel8×2 → PC if C==0 PC+2+rel8×2 → PC if N==1 PC+2+rel8×2 → PC if N==0 PC+2+rel8×2 → PC if V==1 PC+2+rel8×2 → PC if V==0 PC+2+rel8×2 → PC if V xor N==1 PC+2+rel8×2 → PC if V xor N==0 PC+2+rel8×2 → PC if (V xor N) or Z==1 PC+2+rel8×2 → PC if (V xor N) or Z==0 PC+2+rel8×2 → PC if C or Z==1 PC+2+rel8×2 → PC if C or Z==0	

Notes:

- The field "rel8" in the TYPE-D instruction format and the field "rel11" in the TYPE-F format have the following relation to the values "label9" and "label12" in assembly notation.

  label9 → rel8=(label9 – PC – 2)/2

  label12 → rel11=(label12 – PC – 2)/2
- The value "2/1" in the cycle(CYC) column indicates "2" cycles if branching, "1" if not branching.
- It is necessary to set the S flag to "0" for RETI execution.

Table A.2-12 Delayed Branching Instructions (20 Instructions)

Mnemonic	Format	OP	CYC	FLAG NZVC	Operation	Remarks
JMP:D @Ri	E	9F-0	1	– – – –	Ri → PC	
CALL:D label12	F	D8	1	– – – –	PC+4 → RP ,PC+2+rel11×2 → PC	
CALL:D @Ri	E	9F-1	1	– – – –	PC+4 → RP, Ri → PC	
RET:D	E	9F-2	1	– – – –	RP → PC	Return
BNO:D label9	D	F1	1	– – – –	No branch	
BRA:D label9	D	F0	1	– – – –	PC+2+rel8×2 → PC	
BEQ:D label9	D	F2	1	– – – –	PC+2+rel8×2 → PC if Z==1	
BNE:D label9	D	F3	1	– – – –	PC+2+rel8×2 → PC if Z==0	
BC:D label9	D	F4	1	– – – –	PC+2+rel8×2 → PC if C==1	
BNC:D label9	D	F5	1	– – – –	PC+2+rel8×2 → PC if C==0	
BN:D label9	D	F6	1	– – – –	PC+2+rel8×2 → PC if N==1	
BP:D label9	D	F7	1	– – – –	PC+2+rel8×2 → PC if N==0	
BV:D label9	D	F8	1	– – – –	PC+2+rel8×2 → PC if V==1	
BNV:D label9	D	F9	1	– – – –	PC+2+rel8×2 → PC if V==0	
BLT:D label9	D	FA	1	– – – –	PC+2+rel8×2 → PC if V xor N==1	
BGE:D label9	D	FB	1	– – – –	PC+2+rel8×2 → PC if V xor N==0	
BLE:D label9	D	FC	1	– – – –	PC+2+rel8×2 → PC if (V xor N) or Z==1	
BGT:D label9	D	FD	1	– – – –	PC+2+rel8×2 → PC if (V xor N) or Z==0	
BLS:D label9	D	FE	1	– – – –	PC+2+rel8×2 → PC if C or Z==1	
BHI:D label9	D	FF	1	– – – –	PC+2+rel8×2 → PC if C or Z==0	

Notes:

- The field "rel8" in the TYPE-D instruction format and the field "rel11" in the TYPE-F format have the following relation to the values "label9" and "label12" in assembly notation.

  label9 → rel8=(label9 – PC – 2)/2

  label12 → rel11=(label12 – PC – 2)/2
- Delayed branching instructions are always executed after the following instruction (the delay slot).
- In order to occupy a delay slot, an instruction must satisfy either of the following conditions. Any other instructions used in this position may not be executed according to definition.
  - Instructions other than branching instructions, with the cycle (CYC) column showing the value "1".
  - Instructions with the cycle (CYC) column showing the value "a", "b", "c", or "d".

Table A.2-13 Direct Addressing Instructions (14 Instructions)

Mnemonic	Format	OP	CYC	FLAG NZVC	Operation	Remarks
DMOV @dir10, R13	D	08	b	– – – –	(dir10) → R13	Word
DMOV R13, @dir10	D	18	a	– – – –	R13 → (dir10)	Word
DMOV @dir10, @R13+	D	0C	2a	– – – –	(dir10) → (R13),R13+=4	Word
DMOV @R13+, @dir10	D	1C	2a	– – – –	(R13) → (dir10),R13+=4	Word
DMOV @dir10, @–R15	D	0B	2a	– – – –	R15–=4,(dir10) → (R15)	Word
DMOV @R15+, @dir10	D	1B	2a	– – – –	(R15) → (dir10),R15+=4	Word
DMOVH @dir9, R13	D	09	b	– – – –	(dir9) → R13	Half-word
DMOVH R13, @dir9	D	19	a	– – – –	R13 → (dir9)	Half-word
DMOVH @dir9, @R13+	D	0D	2a	– – – –	(dir9) → (R13),R13+=2	Half-word
DMOVH @R13+, @dir9	D	1D	2a	– – – –	(R13) → (dir9),R13+=2	Half-word
DMOVB @dir8, R13	D	0A	b	– – – –	(dir8) → R13	Byte
DMOVB R13, @dir8	D	1A	a	– – – –	R13 → (dir8)	Byte
DMOVB @dir8, @R13+	D	0E	2a	– – – –	(dir8) → (R13),R13++	Byte
DMOVB @R13+, @dir8	D	1E	2a	– – – –	(R13) → (dir8),R13++	Byte

Note:

The field "dir" in the TYPE-D instruction format has the following relation to the values "dir8" to "dir10" in assembly notation.

- dir8 → dir=dir8
- dir9 → dir=dir9 >> 1
- dir10 → dir=dir10 >> 2

Table A.2-14 Resource Instructions (2 Instructions)

Mnemonic	Format	OP	CYC	FLAG NZVC	Operation	Remarks
LDRES @Ri+, #u4	C	BC	a	– – – –	(Ri) → resource u4 Ri + =4	u4: Channel number
STRES #u4, @Ri+	C	BD	a	– – – –	Resource u4 → (Ri) Ri + =4	u4: Channel number

Table A.2-15 Coprocessor Instructions (4 Instructions)

Mnemonic	Format	OP	CYC	FLAG NZVC	Operation	Remarks
COPOP #u4, #CC, CRj, CRi	E	9F-C	2+a	– – – –	Designates operation	
COPLD #u4, #CC, Rj, CRi	E	9F-D	1+2a	– – – –	Rj → CRi	
COPST #u4, #CC, CRj, Ri	E	9F-E	1+2a	– – – –	CRj → Ri	
COPSV #u4, #CC, CRj, Ri	E	9F-F	1+2a	– – – –	CRj → Ri	No error trap generated

Table A.2-16 Other Instructions (16 Instructions)

Mnemonic	Format	OP	CYC	FLAG NZVC	Operation	RMW	Remarks
NOP	E	9F-A	1	– – – –	No change	-	
ANDCCR #u8 ORCCR #u8	D D	83 93	c c	CCCC CCCC	CCR and u8 → CCR CCR or u8 → CCR	- -	
STILM #u8	D	87	1	– – – –	u8 → ILM	-	Sets "ILM" immediate value
ADDSP #s10	D	A3	1	– – – –	R15 += s10	-	"ADD SP" instruction
EXTSB Ri EXTUB Ri EXTSH Ri EXTUH Ri	E E E E	97-8 97-9 97-A 97-B	1 1 1 1	– – – – – – – – – – – – – – – –	Sign extension 8 → 32bit Zero extension 8 → 32bit Sign extension 16 → 32bit Zero extension 16 → 32bit	- - - -	
LDM0 (reglist) LDM1 (reglist)	D D	8C 8D	See notes below.	– – – – – – – –	(R15) → reglist, increment R15 (R15) → reglist, increment R15	- -	Load multiple R0 to R7 Load multiple R8 to R15
STM0 (reglist) STM1 (reglist)	D D	8E 8F	See notes below.	– – – – – – – –	Decrement R15 reglist → (R15) Decrement R15 reglist → (R15)	- -	Store multiple R0 to R7 Store multiple R8 to R15
ENTER #u10	D	0F	1+a	– – – –	R14 → (R15 – 4), R15 – 4 → R14, R15 – u10 → R15	-	Function entry processing
LEAVE	E	9F-9	b	– – – –	R14 + 4 → R15, (R15 – 4) → R14	-	Function exit processing
XCHB @Rj, Ri	A	8A	2a	– – – –	Ri → TEMP (Rj) → Ri TEMP → (Rj)	❍	Byte data for semaphore processing

Notes:

- In the "ADD SP" instruction, the field "s8" in the TYPE-D instruction format has the following relation to the value "s10" in assembly notation.

  s10 → s8=s10 >> 2
- In the "ENTER" instruction, the field "u8" in the TYPE-D instruction format has the following relation to the value "u10" in assembly notation.

  u10 → u8=u10 >> 2
- The number of execution cycles for the "LDM0" (reglist) and "LDM1" (reglist) instructions is:
  a × (n – 1) + b + 1 cycles, where "n" is the number of registers designated.
- The number of execution cycles for the "STM0" (reglist) and "STM1" (reglist) instructions is:
  a × n+1 cycles, where "n" is the number of registers designated.

APPENDIX B Instruction Maps

This appendix presents FR family instruction map and "E" format.

B.1 Instruction Map

This section shows instruction maps for FR family CPU.

■ Instruction Map

Table B.1-1 Instruction Map

Higher 4 bits (columns) / Lower 4 bits (rows)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	LD @(R13,Rj), Ri	ST Ri, @(R13,Rj)	LD @(R14, disp10),Ri	ST Ri,@ (R14, disp10)	LDUH @(R14, disp9),Ri	STH Ri,@(R14, disp9)	LDUB @(R14, disp8),Ri	STB Ri,@(R14, disp8)	BANDL #u4,@Ri	BORL #u4,@Ri	ADDN #i4,Ri	LSR #u4,Ri	LDI:8 #i8,Ri	CALL label12	BRA label9	BRA:D label9
1	LDUH @(R13,Rj), Ri	STH Ri, @(R13,Rj)							BANDH #u4,@Ri	BORH #u4,@Ri	ADDN2 #i4,Ri	LSR2 #u4,Ri			BNO label9	BNO:D label9
2	LDUB @(R13,Rj), Ri	STB Ri, @(R13,Rj)							AND Rj,Ri	OR Rj,Ri	ADDN Rj,Ri	LSR Rj,Ri			BEQ label9	BEQ:D label9
3	LD @(R15, udisp6),Ri	ST Ri, @(R15,ud6)							ANDCCR #u8	ORCCR #u8	ADDSP #s10	MOV Ri,Rs			BNE label9	BNE:D label9
4	LD @Rj,Ri	ST Ri,@Rj							AND Rj,@Ri	OR Rj,@Ri	ADD #i4,Ri	LSL #u4,Ri			BC label9	BC:D label9
5	LDUH @Rj,Ri	STH Ri,@Rj							ANDH Rj,@Ri	ORH Rj,@Ri	ADD2 #i4,Ri	LSL2 #u4,Ri			BNC label9	BNC:D label9
6	LDUB @Rj,Ri	STB Ri,@Rj							ANDB Rj,@Ri	ORB Rj,@Ri	ADD Rj,Ri	LSL Rj,Ri			BN label9	BN:D label9
7	E format	E format							STILM #u8	E format	ADDC Rj,Ri	MOV Rs,Ri			BP label9	BP:D label9
8	DMOV @d10,R13	DMOV R13,@d10							BTSTL #u4,@Ri	BEORL #u4,@Ri	CMP #i4,Ri	ASR #u4,Ri		CALL:D label12	BV label9	BV:D label9
9	DMOVH @d9, R13	DMOVH R13, @d9							BTSTH #u4,@Ri	BEORH #u4,@Ri	CMP2 #i4,Ri	ASR2 #u4,Ri			BNV label9	BNV:D label9
A	DMOVB @d8, R13	DMOVB R13, @d8							XCHB @Rj,Ri	EOR Rj,Ri	CMP Rj,Ri	ASR Rj,Ri			BLT label9	BLT:D label9
B	DMOV @d10,@−R15	DMOV @R15+,@d10							MOV Rj,Ri	LD:20 #i20,Ri	MULU Rj,Ri	MULUH Rj,Ri			BGE label9	BGE:D label9
C	DMOV @d10,@R13+	DMOV @R13+,@d10							LDM0 (reglist)	EOR Rj,@Ri	SUB Rj,Ri	LDRES @Ri+,#u4			BLE label9	BLE:D label9
D	DMOVH @d9, @R13+	DMOVH @R13+, @d9							LDM1 (reglist)	EORH Rj,@Ri	SUBC Rj,Ri	STRES #u4,@Ri+			BGT label9	BGT:D label9
E	DMOVB @d8, @R13+	DMOVB @R13+, @d8							STM0 (reglist)	EORB Rj,@Ri	SUBN Rj,Ri				BLS label9	BLS:D label9
F	ENTER #u10	INT #u8							STM1 (reglist)	E format	MUL Rj,Ri	MULH Rj,Ri			BHI label9	BHI:D label9

B.2 "E" Format

This section shows "E" format for FR family CPU.

■ "E" Format

Table B.2-1 "E" Format

Higher 8 bits

Lower 4 bits

	07	17	97	9F
0	LD @R15+,Ri	ST Ri,@–R15	JMP @Ri	JMP:D @Ri
1	MOV Ri,PS	MOV PS,Ri	CALL @Ri	CALL:D @Ri
2	–	–	RET	RET:D
3	–	–	RETI	INTE
4	–	–	DIV0S Ri	–
5	–	–	DIV0U Ri	–
6	–	–	DIV1 Ri	DIV3
7	–	–	DIV2 Ri	DIV4S
8	LD @R15+,Rs	ST Rs,@–R15	EXTSB Ri	LDI:32 #i32,Ri
9	LD @R15+,PS	ST PS,@–R15	EXTUB Ri	LEAVE
A	–	–	EXTSH Ri	NOP
B	–	–	EXTUH Ri	–
C	–	–	–	COPOP #u4, #CC,CRj,CRi
D	–	–	–	COPLD #u4, #CC,Rj,CRi
E	–	–	–	COPST #u4, #CC,CRj,Ri
F	–	–	–	COPSV #u4, #CC,CRj,Ri

-: Undefined

INDEX

The index follows on the next page.
This is listed in alphabetical order.

Index

M

S

CM71-00101-5E

FUJITSU SEMICONDUCTOR • CONTROLLER MANUAL

FR Family

32-BIT MICROCONTROLLER

INSTRUCTION MANUAL

December 2007 the fifth edition

Published **FUJITSU LIMITED** Electronic Devices

Edited Strategic Business Development Dept